# Introduzione a Java

Andrea Gini

## Le basi della programmazione

Imparare a programmare è un'impresa tutt'altro che facile. Chi non ha mai lavorato con un computer potrebbe restare disorientato dalla terminologia specialistica, o far fatica ad apprendere l'uso corretto dei vari strumenti: compilatori, debugger, editor e così via.

La difficoltà maggiore, per chi si trova alla prima esperienza, è senza dubbio quella di riuscire a capire come opera una macchina. Ma una volta superati i primi ostacoli, cresce l'entusiasmo per un'attività che si presenta stimolante e ricca di sfide, una professione in cui la creatività dell'uomo è l'unico strumento che permette di sopperire alla totale assenza di creatività della macchina.

## Programma e linguaggio di programmazione

Java, l'oggetto di studio di questo libro, è un linguaggio di programmazione: per poterne intraprendere lo studio, è bene chiarire cosa sia un programma. Un programma è una sequenza di istruzioni che spiegano a un computer i passi necessari a svolgere un determinato compito. Normalmente, il compito consiste nell'elaborare i dati introdotti dall'operatore mediante una periferica di input, come la tastiera, il mouse o il disco, e nel produrre un determinato output, come un report a video, un file su disco o un foglio stampato. I linguaggi di programmazione, a differenza del linguaggio naturale, hanno un lessico piuttosto semplice con vincoli sintattici molto rigidi: tali caratteristiche sono necessarie per togliere ai linguaggi di programmazione l'ambiguità caratteristica dei linguaggi naturali, e permetterne un'interpretazione univoca.

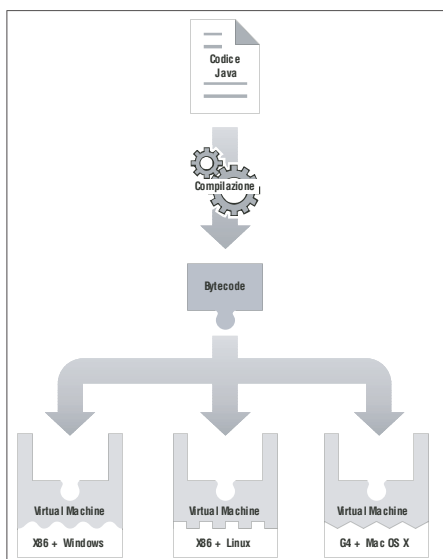
Formalmente un programma è un semplice file di testo, detto "file sorgente", scritto dal programmatore in un determinato linguaggio di programmazione, in questo caso Java, ricorrendo a un apposito ambiente di sviluppo, che nel caso di Java può anche essere un semplice editor di testo come Notepad, Emacs o Vi.

Si noti che l'operatore svolge un ruolo differente a seconda che sia il programmatore o l'utente di un programma. Il programmatore è tenuto a conoscere il funzionamento interno del programma che sta realizzando, mentre l'utente si limita a seguire le istruzioni a video, a inserire i dati quando richiesti e a leggere i risultati. Ogni programmatore è tenuto a seguire due linee guida fondamentali: la prima è scrivere programmi chiari e facili da capire, in modo da permettere a eventuali collaboratori di interpretare il codice senza difficoltà; la seconda è realizzare programmi facili da usare, per rendere la vita più semplice agli utenti.

## Compilatore e Virtual Machine

Il codice sorgente di un programma non può essere eseguito direttamente da un calcolatore: prima deve essere elaborato da uno strumento, detto compilatore, che traduce il file sorgente in un file eseguibile in linguaggio macchina. Tale linguaggio, sebbene sia incomprensibile per un essere umano, è ideale per il calcolatore, che lo esegue direttamente o tramite un apposito interprete.

Il linguaggio Java appartiene alla categoria dei linguaggi interpretati: il compilatore, anziché generare direttamente linguaggio macchina, produce un file eseguibile in un formato detto bytecode, che può essere eseguito da qualsiasi computer grazie a uno speciale interprete, chiamato Java Virtual Machine o JVM. La traduzione dei programmi Java in bytecode garantisce la piena portabilità del codice eseguibile che, al contrario di quanto avviene con altri linguaggi, può essere eseguito su qualsiasi computer che disponga di una JVM, indipendentemente dall'architettura della macchina o dal sistema operativo.

**Figura 1 –** *Un programma Java compilato in bytecode può girare su qualsiasi computer che disponga di una Java Virtual Machine.*

La natura interpretata di Java è allo stesso tempo il suo punto di forza e il suo limite: se da una parte essa garantisce la piena portabilità del file eseguibile e fornisce al programmatore servizi ad alto livello (come la gestione automatica della memoria), dall'altra risulta penalizzante sul piano delle prestazioni. D'altra parte, le attuali Virtual Machine sono in grado di ottimizzare automaticamente le performance di esecuzione del bytecode grazie a un processo di ricompilazione dinamica, che rende il linguaggio Java efficiente quanto altri linguaggi non interpretati come C++. In secondo luogo, il livello di performance raggiunto dalle macchine odierne, dotate di processori che superano in velocità la soglia dei Gigahertz, è tale da rendere il divario di prestazioni pienamente accettabile nella maggior parte dei contesti applicativi.

# Il primo approccio con la programmazione

Lo studio di Java può essere suddiviso in tre tappe:

1. Esame dei costrutti del linguaggio.

2. Filosofia della programmazione a oggetti.

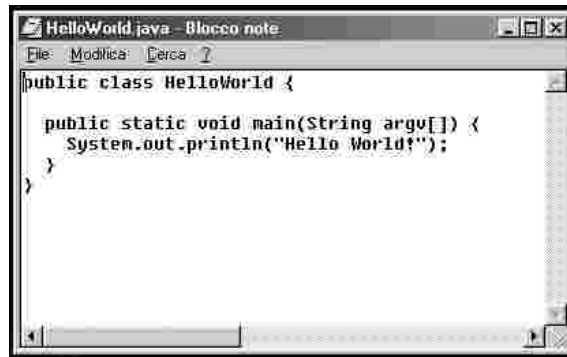
3. Introduzione all'uso dei principali strumenti di libreria.

Chi si avvicina a Java dopo anni di esperienza su linguaggi procedurali (come C o Pascal), di programmazione batch in Cobol o di programmazione host su mainframe, può incontrare altrettante difficoltà di chi si trova alla prima esperienza in assoluto. Prima di iniziare il percorso di studio, è bene ripassare l'insieme dei concetti che si trovano alla base di qualsiasi linguaggio di programmazione.

Per prendere confidenza con i concetti di codice sorgente, compilatore e interprete, è utile portare a termine un'esercitazione. La carriera del programmatore, in ossequio a una tradizione oramai consolidata, prende il via con un gioioso saluto: "Hello World!" Hello World è senza dubbio uno dei programmi più longevi della storia dell'informatica: tradotto in decine di linguaggi differenti, svolge da anni un ruolo da anfitrione, guidando l'aspirante programmatore alla scoperta dei primi segreti dello sviluppo software. Se lo sconosciuto autore avesse avuto la lungimiranza di registrarne il brevetto, certamente avrebbe fatto fortuna.

Il testo del programma in versione Java è il seguente:

```
public class HelloWorld {

  public static void main(String argv[]) {
    System.out.println("Hello World!");
  }
}
```

**Figura 2 –** *Il semplice Notepad consente di scrivere il primo programma.*

Il primo compito da svolgere è copiare il testo del programma all'interno del Notepad, rispettando diligentemente le spaziature e la distinzione tra maiuscole e minuscole. Una volta terminata la trascrizione, è necessario salvare il file con il nome "HelloWorld.java", in una posizione nota del disco come la radice del disco C.

Il file "HelloWorld.java" appena creato è un esempio di file sorgente; per poterlo eseguire è necessario innanzitutto tradurlo in bytecode con il compilatore. Il compilatore standard per Java è un programma a riga di comando, privo cioè di interfaccia grafica. Per poterlo eseguire, è necessario aprire la console di comandi MS Dos, che si trova normalmente sotto la voce Programmi del menu Start in Windows 95/98 o può essere richiamata digitando "cmd" nel pannello Esegui sotto Windows 2000/XP.

Ora è necessario portarsi sulla directory nella quale il file "HelloWorld.java" è stato salvato: se, come consigliato, fosse stato posto nella root del disco C, bisogna digitare il comando "cd c:\", e premere invio. A questo punto, è necessario digitare la seguente riga:

```
javac HelloWorld.java
```

Dopo pochi attimi ricomparirà il cursore, segnalando che la compilazione è andata a buon fine. Nella directory di lavoro, oltre al file "HelloWorld.java", ci sarà ora anche un file "HelloWorld.class" contenente il programma in bytecode.

**Figura 3 –** *Il file sorgente "HelloWorld.java" e il file "HelloWorld.class", risultato della compilazione.*

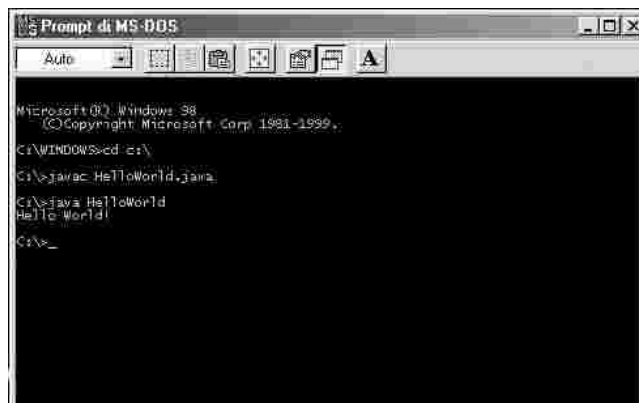
Per eseguire il programma, ora bisogna digitare:

```
java HelloWorld
```

Il sistema si metterà al lavoro, e dopo qualche istante, sulla console apparirà la scritta:

```
HelloWorld
```

Benvenuti nel mondo della programmazione!

**Figura 4 –** *Esempio di utilizzo dei tool Javac e Java del JDK.*

Le istruzioni presenti in questo paragrafo sono valide per la piattaforma Windows. L'utente Unix o Linux, sicuramente più esperto di ambienti a riga di comando, non dovrebbe incontrare particolari difficoltà nell'adattare le istruzioni al proprio caso mediante piccole modifiche.

Chi avesse difficoltà a lavorare con editor di testo e strumenti a riga di comando, può ricorrere a uno dei numerosi ambienti di sviluppo integrati disponibili per Java. Un ambiente di sviluppo è uno speciale editor che permette di scrivere, compilare ed eseguire un programma mediante un'apposita interfaccia grafica. In Appendice B si spiega come installare e usare Ginipad, un ambiente di sviluppo gratuito particolarmente adatto a chi si avvicina per la prima volta a Java. Ginipad è stato sviluppato da Andrea Gini nel contesto di un progetto, supervisionato da MokaByte, che ha dato vita anche a questo libro.

# I costrutti della programmazione

Dopo aver visto come si scrive, si compila e si esegue un semplice programma, è il momento di esaminare alcune possibili variazioni. I linguaggi di programmazione assomigliano al linguaggio naturale inglese, ma rispetto a quest'ultimo prevedono vincoli sintattici molto più rigidi, regolati da un insieme di regole formali che prendono il nome di grammatica. Lo studio di queste regole è la chiave che permette a chiunque lo desideri di realizzare in maniera autonoma dei programmi.

## Struttura base di un programma Java

La struttura base di un programma, in questi primi esempi, sarà sempre del tipo:

```
public class NomeProgramma {
  public static void main(String argv[]) {
  }
}
```

Il nome del programma (che compare dopo la parola "class") può essere scelto liberamente dal programmatore: l'unico vincolo è che il file sorgente che lo contiene abbia lo stesso identico nome, con l'estensione ".java" alla fine. Quando si lavora con Java è bene fare attenzione alla distinzione tra lettere maiuscole e minuscole: il nome "HelloWorld.java" è considerato diverso da "helloWorld.java" dal momento che, nel secondo caso, la lettera "h" è minuscola.

Tra la seconda e la terza parentesi graffa, dopo la scritta `public static void main(String argv[])`, è possibile inserire le istruzioni che verranno illustrate nei prossimi paragrafi.

# Commenti

Per rendere più chiaro un programma è possibile aggiungere veri e propri commenti in lingua naturale, avendo cura di seguire la sintassi descritta in seguito. Il compilatore si limiterà a ignorare tali commenti in fase di compilazione.

## Commento su riga singola

Se si desidera aggiungere un commento lungo non più di una riga, è sufficiente anteporre la sequenza "//" subito prima. Tutto quello che segue i caratteri di inizio commento, fino al termine della riga, verrà ignorato dal compilatore:

```
 // la riga seguente stampa "Hello World!" sullo schermo
System.out.println("Hello World!");
```

## Commento su righe multiple

Per i commenti lunghi più di una riga, è sufficiente ricorrere ai marcatori "/*" e "*/" all'inizio e alla fine del testo:

```
/* Tutto quello che è incluso
tra i marcatori di commento multiriga
viene ignorato dal compilatore*/
```

Esistono numerose modalità di impaginazione dei commenti che ne migliorano la leggibilità. Di seguito è presentata la più comune:

```
/*
 * Tutto quello che è incluso
 * tra i marcatori di commento multiriga
 * viene ignorato dal compilatore
 */
```

## Commento Javadoc

Il linguaggio Java include una modalità di commento, detta Javadoc, che può essere utile nella stesura di progr ammi molto lunghi. La caratteristica più importante di Javadoc, caratterizzato dai tag "/**" e "*/", è che permette al programmatore di generare in modo automatico una valida documentazione ipertestuale, che può rivelarsi preziosa in numerosi frangenti. Javadoc consente di associare a ogni elemento del codice sorgente un gran numero di informazioni, come autore, numero di versione, data di introduzione, link a risorse in rete o ad altri elementi di un programma. La trattazione completa di Javadoc esula dagli obiettivi di questo volume.

# Istruzione elementare

L'istruzione è l'unità di base del programma: essa descrive un compito da eseguire come "somma due numeri" o "disegna un cerchio sullo schermo".

L'istruzione alla quarta riga del programma "HelloWorld.java" ha l'effetto di visualizzare sullo schermo la scritta racchiusa tra virgolette. Si può provare a modificare il sorgente del programma, cambiando la frase con qualcosa tipo "Abbasso la Squola :-P" o "Viva (nome della squadra del cuore)". Per mettere in atto le modifiche sarà necessario salvare di nuovo il file su disco e ripetere la compilazione e l'esecuzione.

Ecco un paio di istruzioni che verranno usate nei prossimi esempi:

```
System.out.println("FRASE");
```

stampa su schermo la frase racchiusa tra virgolette, quindi va a capo;

```
System.out.print("FRASE");
```

stampa su schermo la frase racchiusa tra virgolette, senza andare a capo.

## Sequenza

Il modo più semplice per fornire istruzioni a un calcolatore è scrivere un elenco di operazioni da eseguire una di seguito all'altra. L'interprete eseguirà le istruzioni in sequenza, una alla volta, nello stesso ordine in cui sono state elencate nel testo del programma.

Si provi a modificare il programma precedente in questo modo:

```
public class HelloWorld {

  public static void main(String argv[]) {
    System.out.println("Hello World!");
    System.out.println("Every day is a very nice day");
  }
}
```

Le istruzioni di stampa ora sono due, e verranno eseguite in sequenza. È possibile istruire il computer a eseguire una qualsiasi sequenza di istruzioni:

```
System.out.println("Nel mezzo del cammin di nostra vita");
System.out.println("Mi ritrovai per una selva oscura");
System.out.println("Che la diritta via era smarrita");
....
```

Se le istruzioni sono formulate correttamente, il computer le eseguirà una alla volta, nello stesso ordine in cui sono elencate nel codice sorgente.

# Variabili intere

Se il computer si limitasse a stampare un elenco di frasi sullo schermo, la sua utilità sarebbe decisamente limitata. Quello che rende veramente utile il computer è in primo luogo la sua capacità di eseguire calcoli aritmetici con grande velocità e precisione, mantenendo in memoria i risultati temporanei.

Il principale strumento di manipolazione numerica è la variabile: una cella di memoria al cui interno è possibile memorizzare un numero. Il programmatore può ricorrere alle variabili per effettuare calcoli, memorizzare i risultati di espressioni anche molto complesse ed effettuare confronti.

Il linguaggio Java permette di gestire variabili di diversi tipi. Per il momento, verrà illustrato solo l'uso delle variabili intere.

## Dichiarazione

Prima di utilizzare una variabile è necessario eseguire una dichiarazione, in modo da segnalare al computer la necessità di riservare una cella di memoria. Per dichiarare una variabile si usa la parola "int", seguita da un nome scelto dal programmatore e da un carattere di punto e virgola (;):

```
int a;
int operando;
int risultato;
```

Il nome della variabile deve essere composto da una combinazione di lettere e numeri, di lunghezza arbitraria: normalmente viene scelto in base al contesto di utilizzo, in modo da rendere il programma più leggibile.

## Assegnamento

Per memorizzare un valore all'interno di una variabile, si deve effettuare un assegnamento. La sintassi di un assegnamento è data dal nome della variabile, seguito dal carattere uguale (=) e da un'espressione numerica:

```
a = 10;
operando = 15;
```

Nella parte destra di un assegnamento, dopo il carattere =, è possibile inserire un'espressione aritmetica che faccia uso di parentesi e dei normali operatori aritmetici: + per la somma, - per la sottrazione, * per la moltiplicazione, / per la divisione e % per l'operazione di modulo (resto di una divisione intera). In questi casi la variabile a sinistra assumerà il valore dell'espressione a destra. Per esempio, l'istruzione:

```
risultato = ( 10 + 5 ) * 2;
```

assegnerà il valore 30 alla variabile risultato.

La dichiarazione di una variabile e il primo assegnamento (detto anche inizializzazione) possono essere eseguiti in un'unica istruzione, come nel seguente esempio:

```
int a = 10;
```

### Uso delle variabili

Una variabile, una volta inizializzata, si comporta nel programma come il numero che le viene assegnato. Essa può dunque comparire all'interno di espressioni, ed essere a sua volta utilizzata in un assegnamento:

```
a = 10;
b = a * 2;
```

In questo esempio, alla variabile b viene assegnato il valore della variabile a moltiplicato per due: dal momento che a vale 10, b assumerà il valore 20.

Cosa succede se l'espressione nella parte destra di un assegnamento contiene la variabile da assegnare? Per esempio: che senso ha la seguente istruzione?

```
a = a + 2;
```

In un caso come questo, la variabile a assume il valore che si ottiene valutando l'espressione a destra del carattere = in base al valore "precedente" della variabile. Si osservino per esempio le seguenti istruzioni:

```
a = 10;
a = a + 2;
```

La prima istruzione assegna alla variabile a il valore 10. La seconda, invece, assegna il valore 12, dato dalla somma del precedente valore di a, in questo caso 10, con il numero 2.

## Struttura di controllo decisionale: il costrutto if - else

Quando si descrive una sequenza di operazioni, capita talvolta di dover mettere in pratica delle distinzioni. Se si desidera spiegare a un amico come si prepara una festa, si ricorrerà a una serie di istruzioni di questo tipo:

```
comprare patatine;
comprare torta;
comprare piatti;
....
```

Ma se si desidera che la festa sia un successo, è necessario prendere in considerazione alcuni fattori, e agire di conseguenza:

```
se gli invitati sono maggiorenni
  comprare birra
altrimenti
  comprare aranciata
```

Questo modo di operare è chiamato “condizionale”: l’esecutore valuta una condizione e agisce in modo differente a seconda che essa risulti vera o falsa. La formulazione generale del costrutto condizionale in Java è la seguente:

```
if ( condizione ) {
  // Istruzioni da eseguire se la condizione è vera
}
else {
  // Istruzioni da eseguire se la condizione è falsa
}
```

La clausola `else` è opzionale. Se non si deve specificare un percorso alternativo, è possibile ricorrere alla seguente formulazione:

```
if (condizione ) {
  // Istruzioni da eseguire se la condizione è vera
}
```

## Formulazione di una condizione

Una condizione ha normalmente la forma di un confronto tra variabili e numeri, o tra variabili e variabili. Per verificare l’uguaglianza tra due variabili, è necessario utilizzare l’operatore doppio uguale (==), formato da due caratteri = senza spaziature intermedie:

```
if( a == b ) {
  // Istruzioni da eseguire se a è uguale a b
 }
else {
  // Istruzioni da eseguire se a è diversa da b
}
```

L’operatore != è l’inverso del precedente: esso permette di verificare se due valori sono differenti. Altri operatori importanti sono:

- < per valutare la condizione “minore di”;
- <= per “minore o uguale”;
- > per “maggiore di”;
- >= per “maggiore o uguale”.

Riprendendo in esame l'esempio della festa, si può esprimere la condizione con il seguente pseudo-codice:

```
if(età >= 18) {
  comprareBirra();
 }
else {
  comprareAranciata();
}
```

## Struttura di controllo iterativa: l'istruzione while

In una celebre scena del film *Shining*, su un tavolo da pranzo nel soggiorno dell'Overlook Hotel Wendy (l'attrice Shelley Duval) scopre un'intera risma di fogli per macchina da scrivere con sopra stampata un'unica frase, ripetuta ossessivamente, migliaia di volte:

```
Il mattino ha l'oro in bocca
Il mattino ha l'oro in bocca
Il mattino ha l'oro in bocca
Il mattino ha l'oro in bocca
Il mattino ha l'oro in bocca
Il mattino ha l'oro in bocca
```

Questa scoperta fornisce la prova inconfutabile della pazzia del marito, magistralmente interpretato dal bravissimo Jack Nicholson.

Secondo la leggenda, il regista Stanley Kubrick, insoddisfatto dalla bassa qualità degli strumenti di riproduzione fotostatica dell'epoca, pretese che i cinquecento fogli fossero battuti a macchina uno per uno dai suoi fedeli assistenti in quattro lingue diverse, al fine di ottenere il massimo grado di realismo. Al giorno d'oggi le tecnologie di stampa hanno raggiunto la qualità tipografica, e persino Kubrick, di cui era nota la pignoleria, avrebbe acconsentito ad assegnare un simile folle lavoro a un computer mediante un programma di questo tipo:

```
public class Shining {

  public static void main(String argv[]) {
    int i = 0;

    while(i<25000) {
      System.out.println("Il mattino ha l'oro in bocca");
      i = i + 1;
    }
  }
}
```

La parola while, in inglese, significa “fino a che”: lo scopo del while, infatti, è far ripetere al calcolatore un insieme di istruzioni “fino a che” una certa condizione è vera. Nel piccolo programma di esempio, le istruzioni:

```
System.out.println("Il mattino ha l'oro in bocca");
i = i + 1;
```

alla settima e ottava riga vengono ripetute fino a quando il valore di i, posto inizialmente a zero, rimane minore di 25.000. A ogni iterazione, la variabile i viene incrementata di 1: questo incremento fornisce la garanzia che il ciclo terminerà dopo esattamente 25.000 ripetizioni.

## Struttura generale del ciclo while

La struttura generale del while è:

```
while ( condizione ) {
  istruzione1;
  istruzione2;
  ….
  istruzioneN;
}
```

La condizione è un’espressione del tutto simile a quella che correda l’istruzione if. Se si desidera che un ciclo venga ripetuto all’infinito, è sufficiente specificare una condizione sempre vera, tipo:

```
while ( 0 == 0 ) {
  istruzione1;
  istruzione2;
  ….
  istruzioneN;
}
```

Solitamente, comunque, si desidera che il ciclo venga ripetuto un numero prefissato di volte: in questo caso, è necessario che nel gruppo di istruzioni che compongono il ciclo vi sia anche un assegnamento sulla variabile presente nella condizione. Il tipico ciclo ascendente assume la forma:

```
int i = 0;
while(i<=100) {
  istruzione1;
  istruzione2;
  ….
  istruzioneN;
  i = i + 1;
}
```

In questo ciclo, il contatore i viene inizializzato a 0 e incrementato di un'unità alla volta fino a quando non raggiunge il valore 100. È anche possibile realizzare cicli discendenti (che vadano, per esempio, da 100 a 0). Modificando in modo opportuno l'inizializzazione della variabile indice (che ora parte da 100) e la condizione di uscita (i>=0), l'operazione di incremento viene trasformata in decremento:

```
int i = 100;
while(i>=0) {
  istruzione1;
  istruzione2;
  ....
  istruzioneN;
  i = i - 1;
}
```

# Programmi di esempio

Ricorrendo solo ai costrutti presentati nei paragrafi precedenti, verrà ora mostrato come sia già possibile creare alcuni semplici esempi. Per prima cosa, si provi a realizzare un programma che calcoli la somma dei primi 100 numeri interi:

```
public class SommaNumeri {

  public static void main(String argv[]) {
    int i = 0;
    int somma = 0;

    while(i <= 100) {
      somma = somma + i;
      i = i + 1;
    }
    System.out.print("La somma dei primi 100 numeri è ");
    System.out.println(somma);
  }
}
```

In questo esempio vengono utilizzate due variabili: i e somma. La prima viene inizializzata a zero, e viene incrementata di un'unità a ogni iterazione del ciclo. La seconda variabile, anch'essa inizializzata a zero, serve ad accumulare il valore cercato: dopo il primo ciclo essa vale 1, dopo il secondo 3, dopo il terzo 6 e così via. Il ciclo termina non appena i supera il valore 100.

Si vuole ora modificare il programma in modo che calcoli, oltre alla somma dei primi 100 interi, la somma dei primi 50 numeri pari. Un modo per distinguere i numeri pari da quelli dispari è considerare il resto della divisione per due: se è uguale a 0, il numero è pari; in caso contrario è dispari.

```
public class SommaNumeriPari {

  public static void main(String argv[]) {
    int i = 0;
    int somma = 0;
    int sommaPari = 0;
    int resto = 0;

    while(i <= 100) {
      somma = somma + i;
      resto = i % 2;

      if(resto == 0) {
        sommaPari = sommaPari + i;
      }

      i = i + 1;
    }
    System.out.print("La somma dei primi 100 numeri è ");
    System.out.println(somma);
    System.out.print(„La somma dei primi 100 numeri pari è „);
    System.out.println(sommaPari);
  }
}
```

A ogni ciclo viene calcolato il resto della divisione per due del numero contenuto nella variabile i. Se è uguale a zero, il valore di i viene sommato a quello della variabile sommaPari:

```
if(resto == 0) {
  sommaPari = sommaPari + i;
}
```

Si vuole ora apportare un'ulteriore modifica al programma di esempio, per fare in modo che calcoli, oltre alla somma dei primi 100 interi e dei primi 50 numeri pari, la somma dei primi 50 numeri dispari.

Ovviamente è necessario inserire una nuova variabile, sommaDispari, per accumulare il risultato. In secondo luogo, bisogna aggiungere la clausola else all'istruzione if introdotta precedentemente:

```
if(resto == 0) {
  sommaPari = sommaPari + i;
      }
else {
  sommaDispari = sommaDispari + i;
}
```

Le due istruzioni verranno eseguite alternativamente: nei cicli pari viene eseguita la prima, nei cicli dispari la seconda. Ecco il codice completo dell'esempio:

```
public class SommaNumeriPariEDispari {

  public static void main(String argv[]) {
    int i = 0;
    int somma = 0;
    int sommaPari = 0;
    int sommaDispari = 0;
    int resto = 0;

    while(i <= 100) {
      somma = somma + i;
      resto = i % 2;

      if(resto == 0) {
        sommaPari = sommaPari + i;
      }
      else {
        sommaDispari = sommaDispari + i;
      }

      i = i + 1;
    }
    System.out.print("La somma dei primi 100 numeri è ");
    System.out.println(somma);
    System.out.print("La somma dei primi 100 numeri pari è ");
    System.out.println(sommaPari);
    System.out.print("La somma dei primi 100 numeri dispari è ");
    System.out.println(sommaDispari);
  }
}
```

# Cosa si può fare con questi costrutti?

I costrutti descritti finora sono solo una piccola parte di quelli offerti da Java. Tuttavia, essi permettono di realizzare un gran numero di programmi. Dopo aver preso confidenza con gli esempi, si può tentare di realizzare qualcosa di diverso: con un po' di fantasia, è possibile individuare migliaia di possibilità. Ci si può domandare quali siano i limiti di questo mini linguaggio, così semplice e primitivo. Quanti programmi è possibile scrivere? Che tipo di applicazioni si possono realizzare? La risposta, per certi versi sconcertante, è che questo piccolo linguaggio permette di realizzare qualsiasi programma si desideri, come Word, Excel, un compilatore Java o un intero sistema operativo.

Nel 1966 due matematici italiani, Corrado Bohm e Giuseppe Jacopini, dimostrarono formalmente che qualsiasi programma per calcolatore, scritto in un qualsiasi linguaggio di programmazione, poteva essere riscritto usando un linguaggio dotato solamente di variabili intere, assegnamento, ciclo while e costrutto condizionale. In altre parole, un linguaggio del tutto equivalente a quello appena descritto. Il teorema di Bohm Jacopini fu un argomento decisivo nel dibattito sull'abolizione del salto incondizionato goto, un costrutto presente nei linguaggi di allora che rendeva il codice estremamente difficile da capire.

Un'interpretazione superficiale del teorema di Bhom Jacopini sembrerebbe suggerire l'inutilità di ulteriori costrutti nei linguaggi di programmazione. Tale affermazione equivarrebbe ad asserire l'inutilità dell'automobile, dal momento che è possibile raggiungere qualsiasi destinazione anche a piedi. Per quanto questo linguaggio permetta, tecnicamente parlando, di realizzare programmi anche molto complessi, è chiaro che oltre un certo livello l'impresa andrebbe al di là delle capacità umane.

L'importanza del teorema di Bohm Jacopini risiede nel fatto che esso definisce in modo chiaro e rigoroso un insieme di nozioni che costituiscono, a tutti gli effetti, la base di qualsiasi linguaggio di programmazione. Nei prossimi capitoli verranno illustrati i costrutti più avanzati del linguaggio: tipi di dati numerici, vettori, stringhe, metodi, classi e oggetti. Ogni nuovo strumento permetterà di realizzare programmi più brevi e più comprensibili di quelli che si possono scrivere con il semplice linguaggio visto in questo capitolo. La possibilità di creare strumenti nuovi mediante un uso sapiente di quelli esistenti è senza dubbio uno degli aspetti più affascinanti della scienza dei calcolatori, ed è la chiave più importante del progresso e dell'innovazione in campo informatico.

# Manuale pratico di Java:
# dalla teoria alla programmazione

# *Indice generale*

# Prefazione

Esiste, fra chi si occupa di didattica dell'informatica, un accordo pressoché generale su quali concetti siano essenziali per una introduzione alla programmazione: programmi, variabili, istruzioni di assegnamento e di controllo, procedure e quant'altro.

Esiste un accordo quasi altrettanto generale sul fatto che la programmazione orientata agli oggetti e in particolare il linguaggio Java siano ormai essenziali nella formazione di un buon programmatore.

Storicamente la didattica della programmazione è stata impostata utilizzando linguaggi imperativi relativamente semplici, come Pascal o C. La programmazione a oggetti veniva introdotta in una fase successiva, e comportava il passaggio a un linguaggio diverso (Java o C++).

Da qualche tempo si è verificata la possibilità di utilizzare direttamente Java come "primo linguaggio" di programmazione, esaminandone prima gli aspetti elementari e subito dopo quelli più evoluti, legati in particolare al concetto di oggetto. Questo approccio ha dato risultati soddisfacenti, in quanto evita le dispersioni derivanti dall'uso di contesti linguistici differenti. Si è però scontrato con la carenza di testi introduttivi basati su Java.

Questo manuale risponde all'esigenza di introdurre alla programmazione partendo "da zero" e appoggiandosi direttamente su Java. Risponde anche a un'altra esigenza: spiegare i concetti in modo accessibile ma rigoroso, evitando sia le banalizzazioni, sia il ricorso a un eccesso di formalismo. Si tratta di un equilibrio difficile, che sembra essere stato qui raggiunto in modo efficace. In concreto: il manuale ha le dimensioni, la comprensibilità, la precisione adeguate per consentire di acquisire in un tempo ragionevole le conoscenze e le competenze di base sulla programmazione in generale e su Java in particolare.

FRANCESCO TISATO
*Professore Ordinario di Informatica*
*Coordinatore dei Corsi di Studio in Informatica*
*Università degli Studi di Milano-Bicocca*

# Tipi di dati

ANDREA GINI

Nell'introduzione è stato introdotto il concetto di assegnamento su variabile intera. Il linguaggio Java offre altri tipi di variabile su cui lavorare: quattro tipi per gli interi, due per i numeri floating point, uno per i caratteri e uno per le variabili booleane. Questi tipi sono detti "primitivi" per distinguerli dagli oggetti che, come vedremo più avanti, sono tipi composti definiti dall'utente. Ogni tipo primitivo è una struttura algebrica composta da un insieme numerico e da un set di operazioni definite su di esso. I tipi primitivi si prestano a un uso intuitivo, che trascende la loro implementazione; esistono tuttavia delle situazioni limite in cui questo approccio non funziona: per evitare comportamenti inattesi è necessario conoscere le proprietà e i limiti di ciascun tipo.

## Tipi interi

L'insieme degli interi, la somma e la sottrazione sono concetti che fanno parte del nostro bagaglio culturale fin dall'età prescolare. I calcolatori hanno una particolare predisposizione per i numeri interi, che possono essere trattati con grande efficienza e precisione assoluta; l'unico problema che può insorgere nel trattare tali numeri è dato dall'estensione del tipo che si utilizza, che non è mai infinita.

Una variabile di tipo `int`, per esempio, può contenere qualsiasi numero intero compreso tra 2.147.483.647 e -2.147.483.648, ossia tutti i numeri rappresentabili con una cella di memoria a 32 bit. Se il valore di una variabile supera la soglia più alta, in questo caso 2.147.483.647, essa ricomincia dal valore opposto, ossia da -2.147.483.648. È importante prestare grande attenzione a questo tipo di limitazioni quando si lavora su un calcolatore.

Il formato int è senza dubbio il tipo primitivo più usato, per ragioni di praticità e di efficienza: i 32 bit forniscono un ottimo compromesso tra l'estensione dell'insieme numerico e le prestazioni su tutte le moderne piattaforme hardware. Esistono comunque altri tre tipi interi, che si distinguono da int per la dimensione massima dei numeri trattabili. Il tipo byte permette di operare su numeri compresi tra -128 e 127, ossia i numeri che è possibile rappresentare con cifre da 8 bit. Malgrado la ridicola estensione dell'insieme sottostante, byte è un tipo di dato estremamente importante, dal momento che è il formato di base nello scambio di dati con le periferiche di input/output, come i dischi o la rete. Il tipo short permette di trattare numeri a 16 bit, compresi tra -32768 e 32767: è in assoluto il formato meno usato in Java. Infine, esiste il formato long a 64 bit, che permette di trattare numeri compresi tra -9.223.327.036.854.775.808 e 9.223.327.036.854.775.807. Malgrado l'estensione da capogiro, esso viene utilizzato esclusivamente nelle circostanze in cui risulti veramente utile: nei calcolatori attuali, quasi tutti a 32 bit, il tipo long viene trattato con minor efficienza rispetto a int. Il tipo long richiede inoltre una certa attenzione in fase di assegnamento: per assegnare valori superiori a quelli consentiti da una cifra a 32 bit, è necessario posporre al numero la lettera L:

```
long number = 4.543.349.547L;
```

**Tabella 1.1 –** *Tipi di dati interi.*

| Tipo | Min | Max |
|---|---:|---:|
| byte | -128 | 127 |
| short | -32768 | 32767 |
| int | -2.147.483.648 | 2.147.483.647 |
| long | -9.223.327.036.854.775.808 | 9.223.327.036.854.775.807 |

# Tipi numerici floating point

Per elaborare numeri con decimali, Java mette a disposizione i tipi floating point (a virgola mobile). Il calcolo a virgola mobile funziona secondo il principio della notazione scientifica, dove un numero viene rappresentato mediante una parte intera, detta mantissa, moltiplicata per un'opportuna potenza (positiva o negativa) di 10:

$0,00000456 = 0,456 * 10^{-6}$
$345\ 675\ 432 = 0,345675432 * 10^{7}$

Questa modalità di rappresentazione offre il vantaggio di permettere la manipolazione di numeri molto più grandi o molto più piccoli di quanto sarebbe consentito dal numero di cifre disponibili, al prezzo di un arrotondamento per eccesso o per difetto:

0,00000000000000000007695439352 ≈ 0,769544 * $10^{-20}$
576 469 830 453 324 239 437 892 544 ≈ 0,57647 * $10^{27}$

I numeri in virgola mobile, malgrado la loro importanza nel calcolo scientifico, non vengono trattati diffusamente nei manuali di base. Le particolari modalità di arrotondamento e di perdita di precisione, caratteristici di questo tipo numerico, richiedono conoscenze matematiche non banali.

Il `float`, a 32 bit, può contenere numeri positivi e negativi compresi tra 1.40129846432481707 * $10^{-45}$ e 3.4282346638528860 * $10^{38}$, mentre il `double` a 64 bit può lavorare su numeri positivi e negativi tra 4.94065655841246544 * $10^{-324}$ e 1.79769313486231570 * $10^{138}$.

**Tabella 1.2 –** *Tipi di dati floating point.*

| Tipo | Min | Max |
|---|---|---|
| Float | 1.40129846432481707 * $10^{-45}$ | 3.4282346638528860 * $10^{38}$ |
| Double | 4.94065655841246544 * $10^{-324}$ | 1.79769313486231570 * $10^{138}$ |

La maggior parte dei calcolatori moderni è ottimizzata per il calcolo floating point a doppia precisione. Le funzioni matematiche presenti nelle librerie Java usano quasi sempre il tipo `double`.

## Assegnamento di variabili floating point

Per l'assegnamento di variabili floating point si ricorre ad una speciale formulazione della notazione esponenziale. Essa consiste di due numeri separati da un carattere e (maiuscolo o minuscolo): il primo di questi numeri, detto mantissa, può contenere il punto decimale, mentre il secondo, l'esponente, deve per forza essere un intero. Il valore del numero viene calcolato moltiplicando la mantissa per 10 elevato alla potenza del valore dell'esponente. Nell'assegnare una variabile float è necessario posporre la lettera F, come negli esempi:

Rappresentazione float: 1.56e3F;
Rappresentazione double: 5.23423e102;

# Booleano

Una variabile booleana può assumere solamente due valori: `true` e `false`, che significano rispettivamente "vero" e "falso". Per operare su valori booleani, è necessario ricorrere a un'algebra particolare, detta algebra booleana, che denota un calcolo caratterizzato da proprietà molto diverse rispetto a quello degli interi. Nonostante l'apparente semplicità, l'algebra booleana ha una potenza enorme: per essere precisi, è l'unica algebra che un calcolatore è in grado di trattare

a livello hardware. La mappatura del calcolo algebrico su quello booleano è un tema estremamente affascinante, che può essere approfondito, a seconda della specifica area di interesse, su un manuale di logica matematica, di informatica teorica o di elettronica digitale; in questa sede verranno approfondite solamente le proprietà elementari dei principali operatori.

## Assegnamento su variabili booleane

Una variabile booleana può assumere solamente i valori true e false; pertanto, il modo più semplice di effettuare un assegnamento consiste nel porre una variabile a uno di questi due valori:

```
boolean a = true;
boolean b = false;
```

Il ricorso agli operatori relazionali ==, !=, >, <, >= e <= permette di assegnare a una variabile booleana il valore di verità di un'espressione. Per esempio:

```
boolean b = (a == 10);
```

assegna a b il valore di verità dell'espressione a == 10, che sarà true se la variabile 'a' contiene il valore 10, false in caso contrario. Le espressioni booleane possono essere combinate tramite gli operatori logici ! (NOT) , & (AND), | (OR) e ^ (XOR). Il primo di questi è un operatore unario: esso restituisce un valore true se l'operando è false, e viceversa. Per esempio:

```
boolean a = false;
boolean b = !a;
```

la variabile b assume il valore opposto ad a, ossia true. L'operando & lavora su due operatori. Esso restituisce true solo se entrambi gli operatori sono true; in tutti gli altri casi restituisce false. L'operatore | lavora su due parametri: esso restituisce true se almeno uno dei due parametri è true (in altre parole, è false solo quando entrambi gli operatori sono false). Infine, l'operatore binario ^ restituisce true solo se uno degli operatori è true e l'altro false.

**Tabella 1.3 –** *Tavole di verità degli operatori booleani.*

| Negazione logica | |
|---|---|
| A | !A |
| false | true |
| false | true |

| And logico | | |
|---|---|---|
| A | B | A&B |
| false | false | false |
| false | true | false |
| true | false | false |
| true | true | true |

| Or logico | | |
|---|---|---|
| A | B | A \| B |
| false | false | false |
| false | true | true |
| true | false | true |
| true | true | true |

| Or esclusivo | | |
|---|---|---|
| A | B | A^B |
| false | false | false |
| false | true | true |
| true | false | true |
| true | true | false |

# Caratteri

Una variabile di tipo char può contenere un carattere in formato Unicode. La codifica Unicode comprende decine di migliaia di caratteri, e include gli alfabeti più diffusi nel mondo. I valori da 0 a 127 corrispondono, per motivi di retro compatibilità, al set di caratteri ASCII. Una variabile char è, a tutti gli effetti, un intero a 16 bit privo di segno: essa può assumere qualsiasi valore tra 0 e 65535. Nell'effettuare assegnamenti, tuttavia, si preferisce ricorrere alle costanti carattere, che si dichiarano racchiudendo un singolo carattere tra apici, come si vede nell'esempio seguente:

```
char carattere1 = 'a';
char carattere2 = 'z';
```

Il simbolo \ ha il ruolo di carattere di escape: grazie a esso è possibile specificare come costante char alcuni caratteri che non sarebbe possibile specificare con la tastiera. La tavola 1.4 presenta l'elenco completo delle sequenze di escape valide.

**Tabella 1.4 –** *Sequenze di escape.*

| Sequenze di escape | |
|---|---|
| \n | nuova linea |
| \r | a capo |
| \f | nuova pagina |
| \' | carattere apice |
| \" | carattere doppio apice |
| \\ | carattere backslash |
| \b | backspace |
| \t | carattere di tabulazione |

# Promozioni e casting

Durante la stesura di un programma capita di dover spostare valori numerici tra variabili di tipo diverso, come tra int e long. Se la variabile di destinazione è più capace di quella di partenza l'operazione, che in questo caso prende il nome di promozione, avviene in modo del tutto trasparente, come negli esempi seguenti:

```
byte b = 100;
short s = b;        // promozione da byte a short
int i = s;          // promozione da short a int
long l = i;         // promozione da int a long
```

È possibile, ancorché sconsigliato, effettuare l'operazione inversa. Un valore definito in una variabile più capiente può essere forzato in una variabile di tipo inferiore, ma nell'operazione possono andare perse delle informazioni. Se si dispone di una variabile intera che contiene un valore di 257 e si prova a forzare un simile valore in una variabile di tipo byte, che per sua natura può contenere valori tra -128 e 127, essa assumerà valore 1. La perdita di informazioni, che dipende dalla particolare modalità di memorizzazione dei valori nei vari tipi, può avere effetti indesiderati, o addirittura drammatici.

Il 4 giugno 1996 a Kourou, nella Guyana Francese, il razzo Ariane 5 si sollevò dalla sua rampa di lancio per un volo di collaudo. Il viaggio inaugurale di Ariane stabilì senza dubbio un primato, dal momento che durò appena quaranta secondi, e terminò con una fragorosa esplosione, che fortunatamente non produsse danni a persone. L'imbarazzante episodio venne trasmesso in mondovisione, con gran dispetto per l'Agenzia Spaziale Europea, che sul progetto Ariane aveva messo in gioco la propria credibilità. Un team di esperti fu incaricato di indagare sul disastro; dopo un'attenta analisi, giunsero alla conclusione che la causa del fallimento era stato un errore software nel sistema di riferimento inerziale. Più precisamente, un numero floating point a 64 bit, relativo alla velocità orizzontale del razzo rispetto alla piattaforma, veniva convertito, mediante un'operazione di casting, in un intero a 16 bit. Non appena il valore superò la fatidica soglia di 32.768, l'operazione cominciò a produrre valori sballati, che mandarono in crisi il sistema di navigazione. Questa circostanza provocò l'attivazione del sistema di autodistruzione, che polverizzò il razzo prima che potesse perdere il controllo e precipitare chissà dove.

Lo sviluppo di Ariane aveva richiesto, nell'arco di un decennio, una spesa complessiva di circa 7 miliardi di dollari. Al momento del lancio, il razzo trasportava quattro satelliti per telecomunicazioni, del valore complessivo di circa 500 milioni di dollari. Una semplice operazione di casting, introdotta, a quanto pare, per discutibili motivi di ottimizzazione, produsse pertanto un danno economico spropositato, oltre a un incalcolabile danno di immagine. A completare il quadro, pare che il carico non fosse neppure stato assicurato, una circostanza che probabilmente fornì nuovi corollari al celebre elenco delle "Leggi di Murphy".

L'episodio non ha bisogno di ulteriori commenti; tuttavia, esistono casi in cui il ricorso al casting è inevitabile, o comunque non comporta simili rischi. Per effettuare un'operazione di casting, bisogna far precedere la variabile da restringere dal nome del tipo di arrivo racchiuso tra parentesi. Le seguenti righe mostrano un esempio inverso al precedente.

```
long l = 100;
int i = (int)l;          // cast da long a int
short s = (short)i;      // cast da int a short
byte b = (byte)s;        // cast da short a byte
```

# Autoincremento e autodecremento

Il linguaggio Java ha ereditato dal C gli operatori di autoincremento, che in molti casi semplificano la sintassi delle espressioni di assegnamento. L'espressione:

```
x = x + 1;
```

può essere riscritta ricorrendo all'operatore ++, come nell'esempio seguente:

```
x++;
```

Allo stesso modo, l'espressione x = x - 1 è equivalente a x--.

Se si desidera effettuare un incremento di valore superiore a 1, si può ricorrere all'operatore +=. L'espressione x = x + 10 può essere riscritta come x += 10.

In modo simile, gli operatori +=, -=, *= e %= permettono di semplificare gli assegnamenti che fanno uso delle altre operazioni aritmetiche.

Gli operatori ++ e -- possono comparire sia prima sia dopo il simbolo di variabile. La differenza tra i due casi è abbastanza sottile: se l'operatore precede la variabile, essa viene dapprima incrementata e poi valutata; quando invece l'operatore segue la variabile, la valutazione avverrà prima dell'operazione di incremento. Nel seguente esempio:

```
x = 10;
y = ++x*2;
```

la variabile x viene incrementata prima che venga calcolato il valore di y, che in definitiva assume il valore 22. Al contrario, nell'esempio:

```
x = 10;
y = x++*2;
```

la variabile x viene prima valutata col valore 10, causando in tal modo l'assegnamento del valore 20 (ossia 10 * 2) alla variabile y; subito dopo, viene incrementata a 11.

# Array

Andrea Gini

Dopo aver introdotto i tipi primitivi, è giunto il momento di analizzare in profondità un altro strumento importantissimo in un linguaggio di programmazione: l'array. Gli array (o vettori) sono collezioni di variabili indicizzate, che permettono di gestire in maniera relativamente semplice grosse porzioni di memoria, e di effettuare su di essa calcoli ripetitivi. Gli array tornano utili in tutte le situazioni in cui si ha l'esigenza di manipolare un gruppo di variabili dello stesso tipo che contengono valori tra loro correlati.

Si immagini di dover scrivere un programma che calcoli la media delle temperature giornaliere; se si dovessero prendere in considerazione 6 misurazioni all'ora, una ogni 10 minuti, servirebbero ben 144 variabili.

```
int temp1 = 15;          // ore 0.00
int temp2 = 16;          // ore 0.10
int temp3 = 16;          // ore 0.20
int temp4 = 16;          // ore 0.30
....
int temp144 = 14;        // ore 11.50
```

Oltre alla scarsa praticità di dover dichiarare 144 variabili, non esiste nessun modo pratico per effettuare calcoli che abbraccino tutto l'insieme dei valori: l'unico sistema per calcolare la media sarebbe quello di realizzare una gigantesca espressione aritmetica del tipo:

```
int media = (temp1 + temp2 + temp3 + .... + temp143 + temp144) / 144;
```

In casi come questo è utile ricorrere a un array, uno strumento concettualmente simile a una tabella, che accomuna sotto un unico nome un insieme di variabili dello stesso tipo:

```
int[] temp = new int[144];
```

La creazione e l'utilizzo degli array presentano alcune differenze rispetto all'impiego delle normali variabili. Nei prossimi paragrafi verranno illustrate una a una.

# Dichiarazione di array

La dichiarazione di un array ha una sintassi un po' più complessa della dichiarazione di variabile semplice. Come per le variabili è necessario indicare un tipo e un nome, con la differenza che, dopo aver specificato il tipo, è necessario posporre una coppia di parentesi quadre:

```
int[] vettoreDiInteri;
```

# Assegnamento

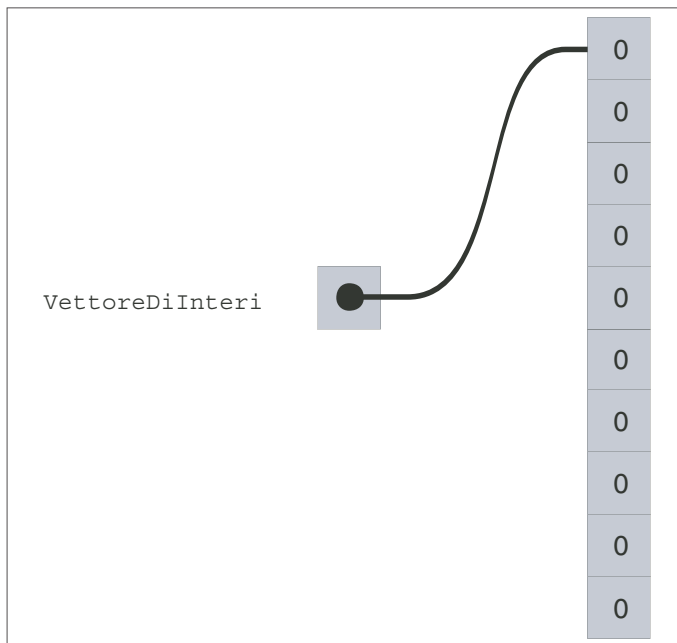
La variabile vettoreDiInteri appena dichiarata non è un vettore, ma solamente un riferimento (in inglese, reference) a un vettore. Il vettore vero e proprio è un'entità separata, che occupa un certo spazio nella memoria e che deve essere creata in modo opportuno. Prima di essere inizializzata, la variabile ha il valore null, una costante che indica che la variabile non referenzia alcun vettore.

**Figura 2.1 –** *La variabile con cui si fa riferimento a un vettore ha inizialmente valore null.*

Per creare un vettore è necessario utilizzare la parola riservata new, come nell'esempio seguente:

```
vettoreDiInteri = new int[10];
```

Il valore tra parentesi quadre è la dimensione del vettore: è possibile specificare un qualsiasi valore intero positivo. Il vettore appena creato è formato da dieci elementi, inizializzati a zero.

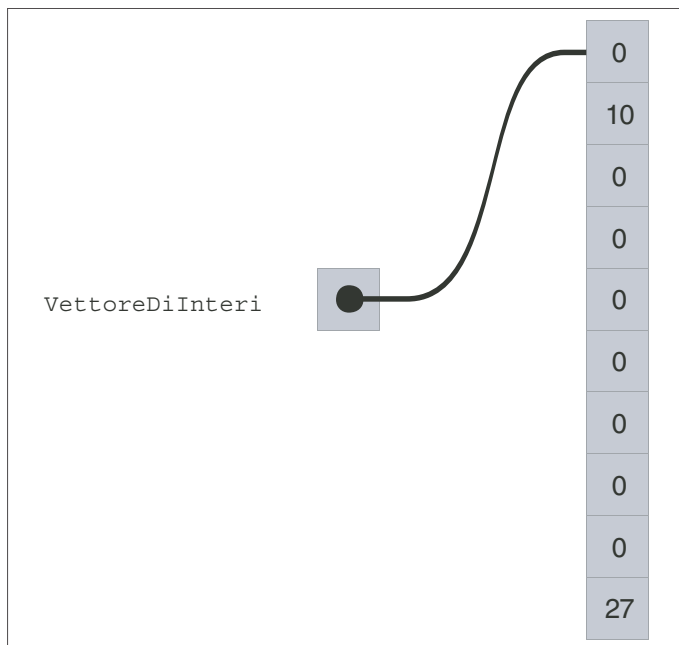
**Figura 2.2 –** *Un vettore è un oggetto di memoria composto da un certo numero di elementi, ognuno dei quali può contenere un valore.*

# Dereferenziazione

La dereferenziazione è l'operazione che permette di assegnare un valore a un elemento del vettore. Per dereferenziare un elemento di un vettore, occorre specificare il nome dell'array seguito dal numero dell'elemento tra parentesi quadre:

```
vettoreDiInteri[1] = 10;
```

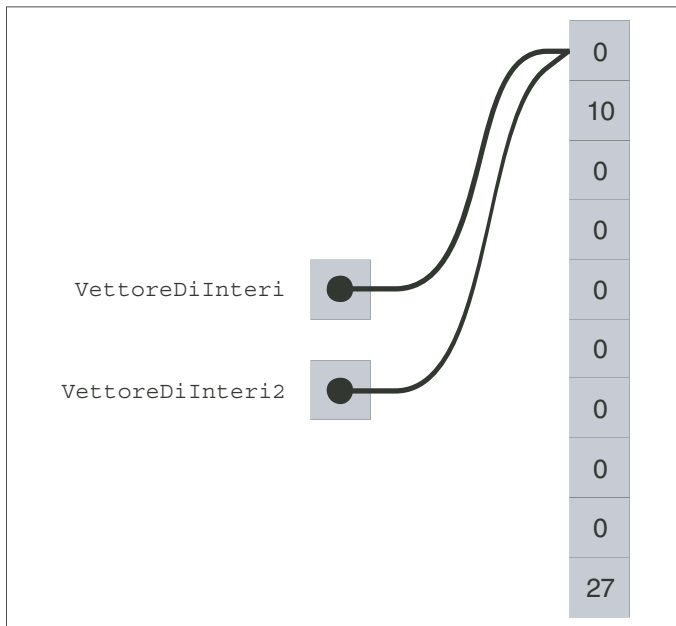
Gli elementi di un vettore si contano a partire da zero: pertanto, se si desidera assegnare il valore 27 al decimo elemento del vettore, è necessario scrivere:

```
vettoreDiInteri[9] = 27;
```

**Figura 2.3 –** *Lo stesso vettore della figura 2.2, dopo aver dereferenziato il secondo e il decimo elemento.*

# Differenza tra assegnamento e dereferenziazione

Quando si lavora su vettori, bisogna avere ben chiara la differenza tra dereferenziazione e assegnamento. L'assegnamento è un'operazione che agisce direttamente sulla variabile, provocandone un cambiamento di valore. La dereferenziazione, invece, è un'operazione indiretta: essa non opera sulla variabile, ma sull'oggetto di memoria puntato da essa. Se si crea un nuovo reference e gli si assegna il valore di un reference che punta a un vettore già esistente, si verifica una situazione in cui due variabili puntano allo stesso vettore, come nell'esempio seguente illustrato in figura 2.4:

```
int[] vettoreDiInteri2;
vettoreDiInteri2 = vettoreDiInteri;
```

**Figura 2.4 –** *Due variabili che fanno riferimento allo stesso vettore.*

In una situazione come questa le operazioni vettoreDiInteri[5] = 10 e vettoreDiInteri[5] = 10 avranno entrambe il risultato di porre a 10 l'elemento numero 5 dell'unico vettore puntato dalle due variabili. Per procedere all'effettiva copia di un vettore, è necessario dapprima creare un vettore delle stesse dimensioni, quindi copiare uno a uno gli elementi del primo nel secondo. Questa operazione può essere eseguita con un ciclo while, come nell'esempio seguente:

```
// crea un vettore e lo inizializza
int[] v1 = new int[5];
v1[0] = 10;
v1[1] = 12;
v1[2] = 14;
v1[3] = 16;
v1[4] = 18;

// crea un vettore della stessa dimensione di v1
int[] v2 = new int[5];
int i = 0;
while(i < 5) {
  // copia il valore della i-esima cella
  // di v1 nella i-esima cella di v2
  v2[i] = v1[i];
}
```

# Inizializzazione automatica di un vettore

Un vettore può essere inizializzato con una serie di valori, in modo simile a come si può fare con le variabili. L'istruzione:

```
int[] vettore = {10,12,14,16,18};
```

equivale alla sequenza di istruzioni:

```
int[] vettore = new int[5];
vettore[0] = 10;
vettore[1] = 12;
vettore[2] = 14;
vettore[3] = 16;
vettore[4] = 18;
```

# Lunghezza di un vettore

I vettori creati con l'operatore `new` hanno esattamente la dimensione specificata nella dichiarazione. Gli indici sono numerati a partire da 0, per cui l'ultimo elemento avrà l'indice pari alla dimensione del vettore meno uno. Per esempio, in un vettore da 10 elementi gli indici sono compresi tra 0 e 9. Il programmatore può essere interessato a conoscere la dimensione di un array a runtime: per questo scopo, ogni vettore dispone di un'apposita costante `length`, accessibile tramite l'operatore '.':

```
int vettoreDiInteri[] = new int[10];
System.out.print("La dimensione del vettore è ");
System.out.println(vettoreDiInteri.length);
```

# Un esempio di manipolazione di vettori

Il vettore è uno strumento potentissimo, che permette di lavorare su porzioni di memoria anche molto grandi usando un numero ridotto di istruzioni. I cicli while permettono di valutare uno a uno gli elementi di un array, e di effettuare qualche tipo di operazione su di essi. Per calcolare la media dei valori contenuti in un ipotetico vettore `vettoreDiInteri`, si può usare un frammento di codice di questo tipo:

```
int i = 0;
int somma = 0;
int media = 0;

while(i<vettoreDiInteri.length) {
```

```
    somma = somma + vettoreDiInteri[i];
    i++;
  }
  media = somma / sommaDiInteri.length;
```

Il seguente esempio permette di togliersi una soddisfazione: quella di scrivere un programma che sfrutti una parte importante della memoria del proprio computer.

Un vettore di byte di dimensione 1024 occupa esattamente un kilobyte (KB) di memoria. Un vettore di int della stessa dimensione ne occupa 4, dal momento che un int è grande 4 byte. Un vettore di interi da 64 megabyte ha una dimensione che può essere calcolata moltiplicando 64 per 1048576 (pari a 1024 al quadrato) e dividendo per quattro. Una volta creato un simile vettore, lo si può riempire di valori casuali scelti tra 0 e 10000; infine, è possibile calcolare la somma di tutti i valori e la relativa media aritmetica. Si noti l'uso di una variabile di tipo long per memorizzare la somma di tutti i numeri: è facile comprendere che una variabile intera non avrebbe la dimensione sufficiente a contenere il risultato.

```
public class MemoryConsumer {

  public static void main(String argv[]) {

    long sum = 0;
    long average = 0;

    // calcola la dimensione del vettore.
    // Se si dispone di poca memoria, impostare
    // un valore più basso nella variabile megaBytes.
    int megaBytes = 64;
    int dim = megaBytes * 1048576 / 4;
    int[] bigArray = new int[dim];


    // riempie il vettore di valori casuali
    int i = 0;
    while ( i < bigArray.length ) {
      bigArray[i] = (int)(32000 * Math.random());
      i++;
    }

    // calcola la somma di tutti i valori
    i = 0;
    while ( i < bigArray.length ) {
      sum = sum + bigArray[i];
      i++;
    }

    // calcola la media
```

```
        average = sum / bigArray.length;

        // stampa i risultati
        System.out.print("La somma di tutti i numeri presenti nel vettore è ");
        System.out.println(sum);
        System.out.print("La media della somma di tutti i numeri presenti nel vettore è ");
        System.out.println(average);
    }
}
```

Il programma deve essere salvato, come di consueto, in un file dal nome "MemoryConsumer.java"; per compilarlo bisogna digitare il comando:

```
javac MemoryConsumer.java
```

mentre per eseguirlo bisogna ricorrere all'istruzione:

```
java MemoryConsumer
```

Dopo qualche istante, il programma stamperà un output del tipo:

```
La somma di tutti i numeri presenti nel vettore è 239999200405
La media della somma di tutti i numeri presenti nel vettore è 15999
```

Per poter eseguire questo programma, è necessario disporre di un computer con almeno 256 MB di RAM. Se non si dispone di memoria sufficiente, il computer segnalerà un errore:

```
java.lang.OutOfMemoryError
        <<no stack trace available>>
Exception in thread "main"
```

In questo caso, si provi a diminuire il valore della variabile 'megaBytes', portandolo per esempio a 32, quindi si provi a compilare ed eseguire nuovamente.

# Vettori multidimensionali

Il linguaggio Java consente di creare vettori bidimensionali, ricorrendo a una sintassi del tipo:

```
int i[][] = new int[10][15];
```

I vettori bidimensionali sono concettualmente simili a una tabella rettangolare, dotata di righe e colonne. Il seguente programma crea una tavola pitagorica in un vettore bidimensionale, quindi la stampa sullo schermo:

```
public class Tabelline2 {

  public static void main(String argv[]) {
    int[][] tabellina = new int[11][11];

    // crea la tabellina in un vettore bidimensionale
    int i = 0;
    int j = 0;
    while(i <= 10) {
      while(j <= 10) {
        int prodotto = i*j;
        tabellina[i][j] = prodotto;
        j = j + 1;
      }
      i = i + 1;
      j = 0;
    }

    // stampa il contenuto del vettore
    i = 0;
    j = 0;
    while(i <= 10) {
      while(j <= 10) {
        int prodotto = i*j;
        System.out.print(tabellina[i][j]);
        System.out.print("\t");
        j = j + 1;
      }
      i = i + 1;
      j = 0;
      System.out.println();
    }
  }
}
```

È possibile definire vettori con un numero qualsiasi di dimensioni:

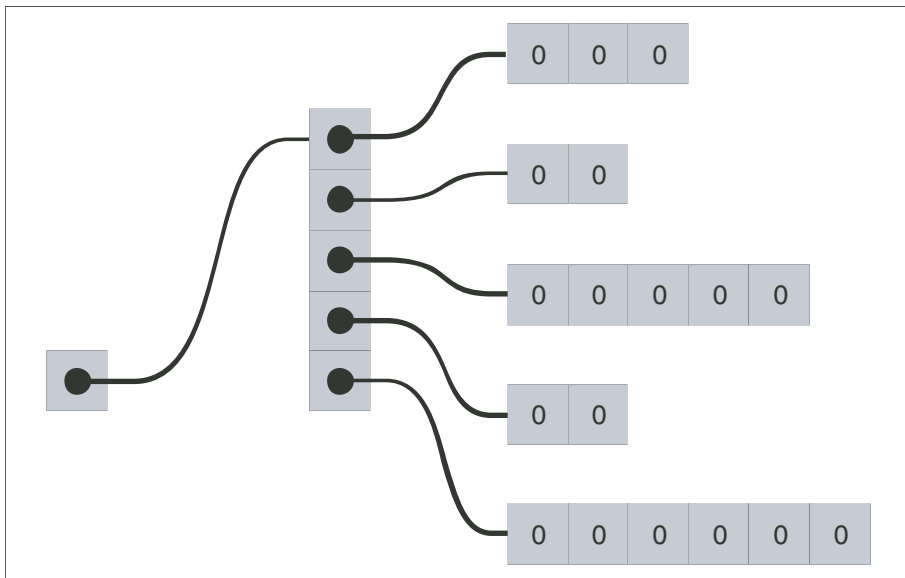
```
int v1[][][] = new int[10][15][5];
int v2[][][][] = new int[10][15][12][5];
```

Tali strutture, in ogni caso, risultano decisamente poco utilizzate.

## Vettori incompleti

I vettori n-dimensionali vengono implementati in Java come array di array. Questa scelta implementativa consente di realizzare tabelle non rettangolari, come nell'esempio seguente:

```
// crea un vettore con una componente incompleta
int tabella[][] = new int[5][];

tabella[0] = new int[3];
tabella[1] = new int[2];
tabella[2] = new int[5];
tabella[3] = new int[2];
tabella[4] = new int[6];
```

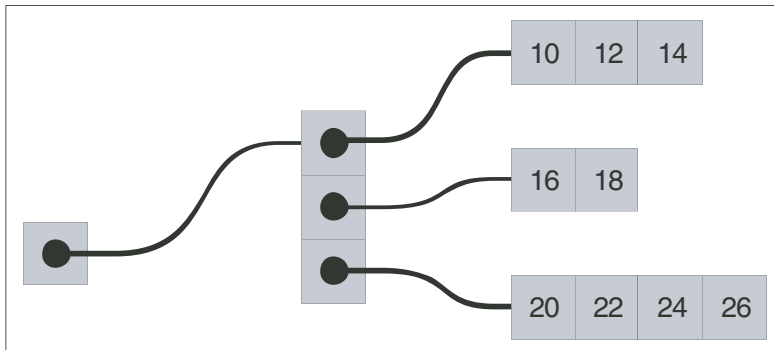
**Figura 2.5 –** *Rappresentazione in memoria di un vettore non rettangolare.*

Per dichiarare un vettore incompleto come quello dell'esempio è necessario specificare la prima componente in fase di creazione, mentre la seconda verrà precisata successivamente creando uno a uno i sotto-vettori. Anche in questo caso ci si trova in presenza di un costrutto scarsamente utilizzato, che tuttavia vale la pena di conoscere.

## Inizializzazione automatica di un vettore multidimensionale

Un vettore può essere inizializzato con una serie di valori, in modo simile a come si può fare con i vettori semplici. Naturalmente è necessario ricorrere a un costrutto un po' più complesso, che tenga conto della particolare struttura di questi vettori. La seguente istruzione, per esempio, crea un vettore a tre componenti in verticale, in cui la prima riga ha tre colonne, la seconda due

e la terza quattro, e contemporaneamente inizializza gli elementi con i valori specificati, come si può vedere in figura 2.6:

```
int[][] vettore = { { 10,12,14} , {16,18} , {20,22,24,26} };
```

**Figura 2.6 –** *Un altro esempio di vettore non rettangolare.*

# Strutture di controllo decisionali

ANDREA GINI

Dopo aver introdotto il concetto di variabile e di array, è giunto il momento di analizzare a fondo i restanti costrutti del linguaggio Java. Come si è già visto nell'introduzione, i costrutti fondamentali di un linguaggio di programmazione sono quelli decisionali e quelli iterativi. Il linguaggio Java prevede tre costrutti decisionali e tre iterativi: in questo capitolo verranno analizzate in profondità le strutture di controllo del primo tipo. I programmatori C troveranno familiari i costrutti di Java: questo infatti riprende la sintassi del C introducendo variazioni minime.

Una struttura di controllo decisionale permette al programmatore di vincolare l'esecuzione di un'istruzione (o di un blocco di istruzioni) a una condizione booleana. Prima di analizzare l'essenza di tali costrutti, è bene chiarire cosa si intenda esattamente con i termini "espressione booleana" e "blocco di istruzioni".

## Condizioni booleane

Una condizione booleana è un'espressione della quale si può dire se sia vera o falsa (in inglese, true o false). Nell'introduzione sono stati già illustrati alcuni semplici esempi, che fanno uso degli operatori di uguaglianza, maggiore e minore. In questa sede vale la pena di approfondire la possibilità di combinare le condizioni booleane mediante gli operatori logici AND, OR, NOT e XOR.

In Java l'operatore AND viene rappresentato dal carattere &. L'AND logico opera su due parametri, e restituisce `true` solamente se entrambi sono `true`. Se si desidera che l'istruzione x = x + 1 venga eseguita solo se il valore della variabile x è maggiore di 10 e contemporaneamente minore di 100 (ossia compreso tra 10 e 100) si può scrivere:

```
if(x >= 10 && x <= 100)
  x = x + 1;
```

Si noti che nell'esempio l'operatore & viene ripetuto due volte: questa variante dell'AND, denominata short circuit, segnala al calcolatore che può interrompere la valutazione dell'espressione non appena sia stata verificato il suo valore di verità, migliorando l'efficienza di esecuzione (per esempio, se durante la valutazione dell'espressione il calcolatore scopre che il primo parametro di una AND è falso, tutta l'espressione risulterà falsa indipendentemente dal valore del secondo parametro).

L'operatore OR, rappresentato in Java con il carattere |, restituisce true se uno o entrambi i parametri sono veri, mentre restituisce false solamente quando entrambi i parametri sono falsi. Pertanto, se si desidera che l'istruzione x = x * 2 venga eseguita solo se x è uguale a 7 o a 8, si dovrà scrivere:

```
if ( x == 7 || x == 8 )
  x = x * 2;
```

Anche in questo caso, si è fatto ricorso all'operatore short circuit || al fine di rendere più efficiente la valutazione: se il primo parametro è vero, l'espressione è vera indipendentemente dal valore del secondo parametro.

L'operatore XOR (OR esclusivo), rappresentato in Java con il carattere ^, restituisce true solo se uno dei due parametri è vero e l'altro falso; se al contrario i parametri sono entrambi veri o entrambi falsi, l'espressione restituisce false. Si noti che non esiste un operatore short circuit per l'OR esclusivo, dal momento che è necessario valutare entrambi i parametri per fornire un valore di verità.

Infine, l'operatore NOT (in Java il carattere !) permette di negare una qualsiasi espressione, restituendo in tal modo un valore true se l'espressione è false e viceversa.

È possibile scrivere espressioni complicate a piacere, combinando tra loro un numero qualsiasi di espressioni più semplici e ricorrendo alle parentesi per rendere esplicite le precedenze. La seguente istruzione azzera la variabile x se il suo valore è compreso tra 10 e 20 o tra 30 e 40, estremi inclusi:

```
if((x>=10 && x <= 20) || (x >= 30 && x <= 40))
  x = 0;
```

# Blocco di istruzioni e variabili locali

Negli esempi visti fino a ora, l'istruzione if è stata usata per vincolare l'esecuzione di un'unica istruzione. Come ci si deve comportare se si desidera vincolare un numero maggiore di istruzioni? In casi come questi si deve definire un blocco, ossia un insieme di istruzioni racchiuso tra parentesi graffe, che il compilatore Java tratta come un'istruzione unica. Pertanto, se si desidera azzerare le variabili x, y e z qualora una di esse superi il valore 100, si può scrivere un frammento di codice del tipo:

```
if ( x >= 100 || y >= 100 || z >= 100 ) {
  // se la condizione è vera, tutte le
  // seguenti istruzioni verranno eseguite
  x = 0;
  y = 0;
  z = 0;
}
```

Per convenzione, quando si apre una parentesi graffa, le righe successive vengono fatte rientrare di un paio di spazi. Questa pratica, che prende comunemente il nome di indentazione, è del tutto arbitraria: niente impedisce di riscrivere il frammento di codice precedente in questo modo:

```
if ( x >= 100 || y >= 100 || z >= 100 )
{ x = 0; y = 0; z = 0; }
```

Come si è già visto in altre occasioni, le convenzioni di impaginazione aiutano a rendere il codice più leggibile, e di conseguenza più facile da correggere o da mantenere.

Una particolarità dei blocchi è che al loro interno è possibile definire variabili locali. Tali variabili hanno una ciclo di vita ridotto, che termina non appena il flusso di esecuzione esce dal blocco.

Pertanto, in un caso come il seguente:

```
if ( x != y ) {
  int t = x;
  x = y;
  y = t;
}
t = 0; // ERRORE!
```

l'ultima istruzione è errata perché fa riferimento alla variabile t, definita all'interno del precedente blocco e che al di fuori di esso ha cessato di esistere.

## if – else

Il costrutto condizionale più usato in Java è l'if, che può essere usato nelle due varianti con o senza else. Il primo tipo, mostrato nel seguente esempio:

```
if ( condizioneBooleana )
  istruzione;
```

esegue l'istruzione se la condizione booleana è vera, mentre prosegue senza fare niente in caso contrario. La variante con l'else ha una forma del tipo:

```
if ( condizioneBooleana )
  istruzione1;
else
  istruzione2;
```

e permette di specificare, oltre all'istruzione da eseguire in caso di successo, anche quella da eseguire in caso di fallimento. Come è stato già spiegato nel paragrafo precedente, se si desidera che venga eseguita più di un'istruzione è necessario ricorrere ai blocchi:

```
if ( condizioneBooleana ) {
  istruzione1a;
  istruzione2a;
  istruzione3a;
}
else {
  istruzione1b;
  istruzione2b;
  istruzione3b;
}
```

## if – else annidati

Il costrutto if può comparire anche all'interno di un altro costrutto if, creando strutture nidificate anche molto complesse. Si osservi un frammento di codice con due if concatenati:

```
if( x >= 0 )
  if( x <= 10 )
    System.out.println("x è compreso tra 0 e 10");
```

Il primo di questi dice "se la variabile x è maggiore o uguale a 0, esegui l'istruzione seguente"; l'istruzione successiva è a sua volta un if che dice "se la variabile x è minore o uguale a cento, esegui l'istruzione successiva": pertanto la terza istruzione verrà eseguita solamente se entrambe le condizioni precedenti risultano vere.

Se si inserisce un else dopo queste istruzioni, a quale dei due if farà riferimento? Nel linguaggio Java, un'istruzione else fa sempre riferimento all'ultimo if della catena (quello più interno). Per sottolineare il concetto, si usa allineare l' else al corrispondente if:

```
if( x >= 0 )
  if( x <= 10 )
    System.out.println("x è compreso tra 0 e 10");
  else
    System.out.println("x è maggiore di 10");
```

Se ora si aggiunge un ulteriore else, esso farà riferimento al primo if:

```
if( x >= 0 )
  if( x <= 10 )
    System.out.println("x è compreso tra 0 e 10");
  else                    // riprende l'istruzione if( x <= 10 )
    System.out.println("x è maggiore di 10");
else                                              // riprende l'istruzione if( x >= 0 )
  System.out.println("x è minore di 0");
```

Come si può fare se si desidera forzare un else a fare riferimento a un if esterno? È possibile rimuovere dall'esempio precedente il primo else, in modo che quello che rimane faccia ancora riferimento all'if più esterno? Per ottenere questo effetto è necessario racchiudere l'if più interno in un blocco; in questo modo, l'if interno verrà trattato come un'istruzione a se stante, priva di else:

```
if( x >= 0 ) {
  if( x <= 10 )
    System.out.println("x è compreso tra 0 e 10");
}
else                              // riprende l'istruzione if( x >= 0 )
  System.out.println("x è minore di 0");
```

È buona norma evitare di ricorrere pesantemente alla nidificazione di istruzioni if, data la confusione che spesso ne segue.

Durante la formulazione di combinazioni condizionali troppo complesse può capitare di commettere errori molto difficili da riconoscere. Con un po' di ragionamento è possibile formulare un'espressione più leggibile ricorrendo agli operatori booleani. Per esempio, il frammento di codice:

```
if( x >= 0 )
  if( x <= 10 )
    System.out.println("x è compreso tra 0 e 10");
```

può essere tranquillamente sostituito dal seguente, in tutto equivalente:

```
if( x >= 0 && x <= 10 )
  System.out.println("x è compreso tra 0 e 10");
```

## if – else concatenati

Un caso più semplice di combinazione condizionale si ha quando si fa seguire un if a un else. In questo caso, la verità dell'espressione booleana presente in ciascun if viene rafforzata dalla condizione di verità di tutte le condizioni if precedenti, premettendo di creare una catena di alternative:

```
if( x <= 0 )
  System.out.println("x è minore o uguale a 0");
```

```
else if( x <= 10)
  System.out.println("x è maggiore di 0 e minore o uguale a 10");
else if ( x <= 20)
  System.out.println("x è maggiore di 10 e minore o uguale a 20");
else
  System.out.println("x è maggiore di 20");
```

Si noti che quando si utilizza una serie di if - else concatenati, l'istruzione else che compare alla fine andrà a coprire tutti i casi non considerati dalle precedenti condizioni.

# Il costrutto switch – case

Il costrutto switch permette di gestire tutte quelle situazioni in cui si deve seguire un percorso differente a seconda del valore di un'espressione.

```
switch (espressione) {
  case val1:
    istruzione_1a;
    istruzione_2a;
    ....
    istruzione_na;
    break;
  case val2:
    istruzione_1b;
    istruzione_2b;
    ....
    istruzione_nb;
    break;
  default:
    istruzione_1default;
    istruzione_2defefault;
    ....
    istruzione_ndefault;
    break;
}
```

L'espressione contenuta tra le parentesi dello switch deve essere di tipo intero (int, byte, short o char); ogni istruzione case lavora su un particolare valore, e fornisce una sequenza di istruzioni da eseguire in quella particolare circostanza. Tale sequenza termina usualmente con l'istruzione break, che forza il computer a uscire dallo switch senza verificare i valori successivi. Nonostante il break sia opzionale, il suo uso è fortemente consigliato.

Dopo aver specificato un numero qualsiasi di case, si può chiudere l'elenco specificando il blocco di default, ossia una sequenza di istruzioni da eseguire se non si è verificato nessuno dei casi precedenti. Il blocco di default è opzionale, e pertanto verrà inserito solamente nelle circostanze nelle quali risulti necessario.

Il seguente esempio stampa uno specifico messaggio se x vale 1, 2 o 3, mentre stamperà un messaggio di default in tutti gli altri casi.

```
switch (x) {
  case 1:
    System.out.println("x è uguale a 1");
    break;
  case 2:
    System.out.println("x è uguale a 2");
    break;
  case 3:
    System.out.println("x è uguale a 3");
    break;
  default:
    System.out.println("x è diverso da 1, 2 e 3");
    break;
}
```

# Espressioni condizionali

All'interno delle espressioni aritmetiche è possibile utilizzare l'operatore ?, che permette di rendere condizionale l'assegnamento di valore. Il costrutto si compone di tre parti: un'espressione booleana, seguita da un carattere ?, e due espressioni generiche separate a loro volta da un carattere :, come mostrato qui di seguito:

```
espressioneBooleana ? espressione1 : espressione2;
```

Quando l'espressione viene valutata, il calcolatore verifica il valore di verità dell'espressione booleana: se risulta vera, l'espressione restituisce il valore della prima espressione; in caso contrario restituisce il valore della seconda. Ecco un esempio di istruzione che assegna alla variabile y il valore assoluto di x, ossia il valore x se x è positivo e -x se x è negativo:

```
y = x < 0 ? -x : x;
```

Ovviamente, è sempre possibile costruire una forma equivalente ricorrendo ad un costrutto if - else:

```
if ( x < 0 )
  y = -x;
else
  y = x;
```

La scelta di una forma o dell'altra deve sempre privilegiare la leggibilità del codice risultante.

# Strutture di controllo iterative

ANDREA GINI

Le strutture di controllo iterative permettono di impostare la ripetizione di un insieme di istruzioni per un determinato numero di volte. In Java esistono tre strutture di controllo di questo tipo: while, for e do – while. Sebbene siano equivalenti, ciascuna di queste tre forme ha un campo di applicazione privilegiato: nei prossimi paragrafi ciascuno di essi verrà approfondito in dettaglio.

## Ciclo while

La struttura fondamentale del ciclo while è:

```
while(condizioneBooleana)
  istruzione;
```

come di consueto, l'istruzione può essere sostituita da un blocco:

```
while(condizioneBooleana) {
  istruzione_1;
  istruzione_2;
  ....
  istruzione_n;
}
```

L'effetto del while è quello di ripetere l'istruzione, o il blocco di istruzioni, fintanto che il valore della condizione booleana è vero. Il blocco all'interno del while deve contenere istruzioni che modifichino la condizione booleana; in caso contrario, il ciclo durerà all'infinito.

L'uso del while richiede alcune precauzioni: nel progetto di un ciclo while, infatti, è necessario considerare con attenzione le condizioni di ingresso e quelle di uscita. Se le condizioni di ingresso sono mal formulate, il computer in fase di esecuzione non entrerà mai in ciclo. Se d'altra parte esiste un errore di logica all'interno del ciclo, si corre il rischio che il computer in fase di esecuzione entri in loop, ossia resti intrappolato all'infinito dentro al ciclo. D'altra parte, ci sono situazioni in cui si desidera creare esplicitamente un ciclo infinito, ricorrendo a formulazioni del tipo:

```
while(true)
istruzione;
```

Così facendo il ciclo proseguirà all'infinito, dal momento che la condizione booleana non diventerà mai falsa.

# Ciclo do – while

Nel ciclo while standard, descritto nel paragrafo precedente, il blocco di istruzioni che costituisce il corpo del while può non essere mai eseguito, qualora le condizioni di ingresso non siano vere al momento di entrare in ciclo. Esistono casi in cui si vuole che tali istruzioni vengano eseguite comunque almeno una volta. Per questo, si può usare il costrutto do - while, la cui struttura è:

```
do
  istruzione;
while(condizioneBooleana)
```

o in alternativa:

```
do {
  istruzione_1;
  istruzione_2;
  ....
  istruzione_n;
}
while(condizioneBooleana)
```

se si desidera che venga eseguito un blocco di istruzioni.

Il costrutto do - while è presente in tutti i linguaggi di programmazione imperativi, a volte con il nome di repeat - until, come nel Pascal. Esso, in ogni caso, risulta poco utilizzato.

# Ciclo for

Per ripetere un blocco di codice un numero prefissato di volte è possibile utilizzare un ciclo while di questo tipo:

```
int i = 0;
while(i < 10) {
  System.out.println(i);
  i++;
}
```

Il linguaggio Java prevede il costrutto for, che permette di gestire esplicitamente questo tipo di situazione, peraltro estremamente comune. La sintassi generale del ciclo for è:

```
for ( inizializzatore ; condizioneBooleana ; incremento)
  istruzione;
```

Anche in questo caso è possibile ricorrere al blocco qualora si desideri far ripetere un maggior numero di istruzioni:

```
for ( inizializzatore ; condizioneBooleana ; incremento) {
  istruzione_1;
  istruzione_2;
  ....
  istruzione_n;
}
```

L'inizializzatore è un'istruzione che viene eseguita una sola volta prima di entrare nel ciclo. La condizione booleana è un'espressione che viene testata, come nel while, prima di ogni ciclo; infine, l'incremento è un'istruzione eseguita automaticamente al termine di ogni iterazione.

Il for permette di riscrivere l'esempio precedente in una forma più compatta:

```
for ( int i = 0 ; i < 10 ; i++ )
  System.out.println(i);
```

Grazie al for, è possibile gestire in modo molto più sicuro tutte le circostanze in cui sia necessario far ripetere un gruppo di istruzioni per un numero prefissato di volte. Come vedremo nei prossimi paragrafi, tuttavia, è possibile usare questo costrutto in modo più sofisticato.

## Uso del for con parametri multipli

È possibile fare in modo che il ciclo for lavori su più di un contatore, replicando l'inizializzatore e l'incremento e separandoli con la virgola. Naturalmente, questo richiede una maggiore attenzione nel progetto del ciclo, la cui complessità cresce enormemente insieme al numero dei parametri. Nel seguente esempio, la variabile i assume i valori tra 0 e 59 con incrementi unitari, mentre la variabile j parte da 1 e viene moltiplicata per 2 a ogni ciclo:

```
public class PotenzeDiDue {
  public static void main(String argv[]) {
```

```
      for ( long i = 0, j = 1; i < 60; i++, j *= 2)
        System.out.println("due elevato alla " + i + " uguale a " + j);
    }
  }
```

Si noti che la variabile j non viene valutata all'interno della condizione booleana: in altre parole, la durata del ciclo viene stabilita unicamente mediante la variabile i. Pertanto è possibile riscrivere l'esempio precedente ricorrendo a un unico contatore tra i parametri del ciclo for, trasferendo l'istruzione di incremento della variabile j all'interno del corpo del ciclo:

```
public class PotenzeDiDue {
  public static void main(String argv[]) {
    long j = 1;
    for ( long i = 0 ; i < 60; i++) {
      System.out.println("due elevato alla " + i + " uguale a " + j);
      j *= 2;
    }
  }
}
```

La scelta tra le due forme va effettuata caso per caso, tenendo presente che spesso è preferibile privilegiare la chiarezza rispetto alla concisione (meglio scrivere più righe che correre il rischio di introdurre errori).

## Omissione di parametri

È possibile lasciare in bianco uno o più parametri del for. Se la variabile che si usa come indice è già stata dichiarata, è possibile omettere l'inizializzazione:

```
int i = 0;
for (  ; i < 10 ; i++)
  System.out.println(i);
```

Se l'istruzione di incremento è presente all'interno del ciclo, si può lasciare in bianco il terzo parametro:

```
for ( int i = 0 ; i < 10 ; ) {
  System.out.println(i);
  i++;
}
```

Infine, se si lascia in bianco la condizione booleana, il ciclo verrà eseguito all'infinito:

```
for ( int i = 0 ; ; i++ )
  System.out.println(i);
```

Ognuno di questi casi può essere gestito con un `while` equivalente, cosa che in genere risulta preferibile perché più facile da capire.

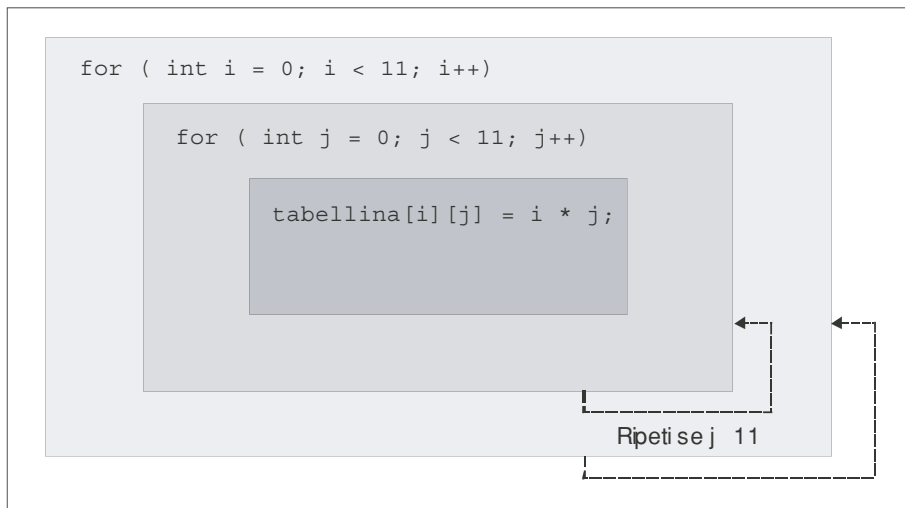
## Cicli nidificati

All'interno di un ciclo è possibile inserire un ciclo ulteriore, ottenendo in tal modo una struttura di controllo nidificata. L'uso di cicli nidificati è indispensabile in molti contesti, come per esempio nel calcolo vettoriale. D'altra parte, l'uso di cicli nidificati richiede una notevole attenzione, dato che proprio in simili situazioni è facile introdurre errori di logica che portano al blocco del programma. L'algoritmo seguente riempie una matrice 11 x 11 con i valori della tavola pitagorica del 10:

```
int[][] tabellina = new int[11][11];
for ( int i = 0; i < 11; i++)     // ciclo esterno
  for ( int j = 0; j < 11; j++)   // ciclo interno
    tabellina[i][j] = i * j;
```

**Figura 4.1 –** *Flusso di esecuzione di un programma con cicli nidificati.*

```
for ( int i = 0; i < 11; i++)
    for ( int j = 0; j < 11; j++)
        tabellina[i][j] = i * j;
```

Ripeti se j 11

Il ciclo più esterno scandisce le righe della matrice; a ogni iterazione viene avviato un ciclo interno che riempie tutti gli elementi della riga. L'istruzione più interna verrà pertanto eseguita 100 volte.

# Uso di break

L'istruzione break, se inserita all'interno di un ciclo while, do - while o for, ha l'effetto di interrompere l'iterazione, e di ricominciare dall'istruzione immediatamente successiva. L'istruzione break consente di inserire condizioni di uscita supplementari, localizzate in punti diversi dall'inizio o dalla fine del ciclo.

Un uso tipico di break è all'interno di algoritmi di ricerca lineare, ossia algoritmi che scandiscono uno a uno gli elementi di un vettore fino a quando non trovano un particolare valore. In prima istanza, una valida soluzione al problema è un algoritmo del tipo:

```
boolean found = false;
for ( int i = 0; i < array.length ; i++ )
  if(array[i] == 101)
    found = true;
if(found)
  System.out.println("Il vettore contiene il valore 101");
```

Dato un array di interi, lo si scandisce dal primo all'ultimo elemento, controllando se uno di essi ha valore 101. Se viene trovato, la variabile booleana found assume il valore true; in caso contrario, al termine del ciclo esso conterrà il valore false. L'ultima istruzione stamperà una scritta se la variabile found è true. Se il vettore è molto grande (si immagini un vettore da 10 o 20 milioni di elementi) e il valore viene trovato dopo appena un centinaio di iterazioni, tutte le iterazioni successive risulteranno inutili. Inserendo un'istruzione break, si può fare in modo che il ciclo duri il minimo necessario per trovare l'elemento, con il vantaggio di ridurre il tempo di esecuzione.

```
boolean found = false;
for ( int i = 0; i < array.length ; i++ )
  if(array[i] == 101) {
    found = true;
    break;  // il ciclo finisce qui
  }
if(found)
  System.out.println("Il vettore contiene il valore 101");
```

# L'istruzione continue

L'istruzione continue ha un effetto simile a quello di break, ma invece di dirottare l'esecuzione al termine del ciclo, la riporta all'inizio e passa all'iterazione successiva. Così, se si desidera realizzare un ciclo che effettui il prodotto di tutti i valori di un vettore, trascurando gli elementi con valore zero, è sufficiente scrivere:

```
long prodotto = 0;
for ( int i = 0; i < array.length ; i++ ) {
```

```
    if(array[i] == 0)
      continue;  // salta direttamente all’iterazione successiva

    prodotto = prodotto * array[i];
  }
```

# Uso di break e continue in cicli nidificati

Quale effetto hanno le istruzioni break e continue in un sistema di cicli nidificati? Esse hanno effetto solo sul ciclo corrente. Se si colloca un’istruzione break in un ciclo esterno, essa provocherà la fine di entrambi i cicli e la prosecuzione del programma, mentre se si pone il break in un ciclo interno, provocherà l’interruzione del solo ciclo interno e la prosecuzione di quello esterno:

```
for ( int i = 0; i < 11; i++) {      // ciclo esterno
  ....
  for ( int j = 0; j < 11; j++) {    // ciclo interno
    ....
    break; // interrompe solo il ciclo interno
  }
  // dopo il break si riparte da qui
  ....
}
```

Un discorso del tutto simile vale per continue:

```
for ( int i = 0; i < 11; i++) {      // ciclo esterno
  ....
  for ( int j = 0; j < 11; j++) {    // ciclo interno
    ....
    continue; // va all’iterazione successiva del ciclo interno
    ....
  }
  ....
}
```

# Uso di break e continue con label

Le istruzioni break e continue possono essere utilizzate in modo più avanzato grazie alle label. Una label è un identificatore seguito da un carattere : e un’istruzione, o un blocco di istruzioni tra parentesi graffe:

```
nome : istruzione;
```

L'istruzione, che in questa circostanza prende il nome di labeled statement (istruzione etichettata), può essere una qualsiasi istruzione Java, compreso un altro ciclo.

L'istruzione break può specificare attraverso una label quale istruzione si intende interrompere. Come si è già visto, l'effetto standard del break è quello di interrompere il ciclo corrente. Il break con etichetta permette di riproporre una variazione dell'esempio precedente in cui il break, sebbene posto nel ciclo più interno, provoca l'interruzione del ciclo più esterno. Per ottenere questo effetto, è necessario etichettare il ciclo più esterno con un nome (cosa che viene fatta nella prima riga) e specificare, dopo il break, l'etichetta assegnata al ciclo più esterno:

```
cicloEsterno :
 for ( int i = 0; i < 11; i++) {     // ciclo esterno
   ....
   for ( int j = 0; j < 11; j++) {   // ciclo interno
     ....
     break cicloEsterno; // interrompe il ciclo esterno
   }
   ....
 }
// dopo il break si riparte da qui
```

Anche l'istruzione continue prevede la variante con label; in questo caso, l'etichetta permette di indicare quale ciclo si intende proseguire:

```
cicloEsterno :
 for ( int i = 0; i < 11; i++) {     // ciclo esterno
   // dopo il continue si prosegue da qui
   ....
   for ( int j = 0; j < 11; j++) {   // ciclo interno
     ....
     continue cicloEsterno; // riprende il ciclo esterno
   }
   ....
 }
```

È inutile sottolineare come l'uso di queste istruzioni tenda a complicare enormemente i programmi laddove, nella maggior parte dei casi, è possibile formulare un costrutto equivalente. Le labeled statement sono un costrutto ereditato dal C, e non capita spesso di incontrarlo nei programmi Java.

# Uso degli oggetti

Andrea Gini

Il paradigma della programmazione a oggetti si è affermato gradualmente a partire dalla metà degli anni Ottanta per le sue caratteristiche di eleganza e naturalezza. La metafora degli oggetti, grazie alla sua analogia con il mondo reale, offre una modalità di rappresentazione dei problemi intuitiva, naturale e a volte persino divertente. Nei prossimi capitoli verranno introdotti in modo graduale i principi base della programmazione in Java, dall'utilizzo di oggetti di libreria alla progettazione di classi fino agli aspetti più sottili, e spesso trascurati o fraintesi, della filosofia degli oggetti.

## La metafora degli oggetti

Durante la progettazione di un sistema informatico, viene spontaneo modellare la struttura e il comportamento dei programmi secondo una metafora che rispecchi l'attività che si intende automatizzare. Le attività del mondo reale sono classificabili come trasformazioni tra oggetti: quando si cucina, tanto per fare un esempio, si trasforma un insieme di ingredienti in una pietanza. Per portare a termine tali compiti, è normale fare ricorso a un certo numero di strumenti: nell'esempio della cucina si può pensare a pentole, posate e fornelli. Nel gergo della programmazione a oggetti non si distingue tra gli oggetti che provocano trasformazioni e quelli che le subiscono (strumenti e ingredienti): gli uni e gli altri vengono descritti col termine generico di *oggetti*. Si esamini ora una procedura che descrive le operazioni necessarie a lessare una patata:

1. Sciacqua la patata sotto al rubinetto fino a quando è pulita.

2. Sbuccia la patata con un coltello.

3. Riempi una pentola con l'acqua del rubinetto.

4. Accendi un fornello nella cucina.

5. Metti la patata nella pentola.

6. Fai bollire la patata fino a che non è cotta.

7. Spegni il fornello.

8. Togli la patata dalla pentola con una forchetta.

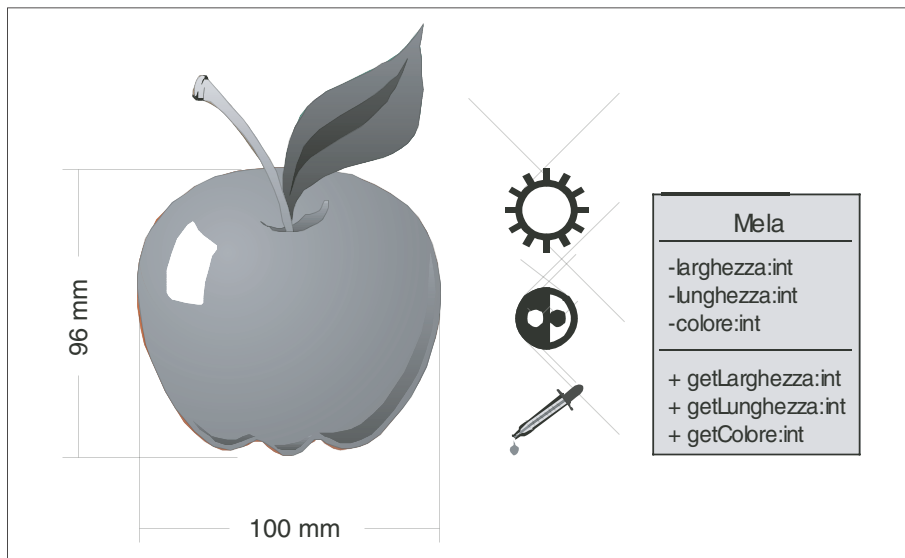
Per portare a termine il compito "cottura di una patata" vengono utilizzati 6 oggetti di uso comune: una patata, un coltello, un rubinetto, una pentola, un fornello e una forchetta. Nel corso dell'operazione, vengono svolte alcune interazioni tra questi oggetti: per sbucciare la patata si utilizza un coltello; per scaldare la pentola si usa un fornello e per cuocere la patata si utilizza la pentola. Le interazioni sono rese possibili dalla natura dell'oggetto stesso: la lama del coltello serve a tagliare, il fuoco a cuocere e così via. Alcuni degli oggetti coinvolti subiscono delle trasformazioni o dei cambiamenti di stato: la patata, inizialmente cruda, è stata dapprima lavata (prima trasformazione), poi sbucciata (seconda trasformazione) e infine cucinata; il fornello, inizialmente spento (primo stato), è stato dapprima acceso (secondo stato) e poi spento di nuovo (terzo stato). D'altra parte, nel descrivere la procedura è stata utilizzato un linguaggio di tipo algoritmico, ricorrendo ai familiari costrutti di sequenza, iterazione e selezione.

Prima di introdurre la metafora degli oggetti nel mondo della programmazione è bene chiarire i concetti di stato e comportamento negli oggetti del mondo reale.

## Lo stato di un oggetto e i suoi attributi

Ogni oggetto del mondo reale può essere descritto mediante un certo numero di attributi, come forma, dimensioni e colore. Qualsiasi qualità misurabile o enumerabile può costituire un attributo. Alcuni attributi, come le dimensioni, sono presenti in tutti gli oggetti; altri invece sono caratteristici di una particolare *classe* di oggetti e non compaiono in altri: "crudo" e "cotto" sono attributi che hanno senso per una patata, ma di certo non per un coltello.

Lo *stato* di un oggetto è l'insieme dei valori dei suoi attributi. Nel corso del suo ciclo di vita, un oggetto può cambiare stato per diverse ragioni: alcuni attributi possono essere modificati dall'utilizzatore tramite un'interazione diretta (il fornello può essere acceso e spento direttamente, utilizzando l'apposita manopola), mentre altri cambiano a seguito di interazioni con altri oggetti (la patata può essere sbucciata solo ricorrendo a un coltello) e altri, infine, cambiano spontaneamente per ragioni interne all'oggetto stesso (una patata matura spontaneamente nel corso del tempo). D'altra parte, ci sono anche attributi immutabili: il peso del coltello, per esempio, non può cambiare, a meno che il coltello non venga distrutto.

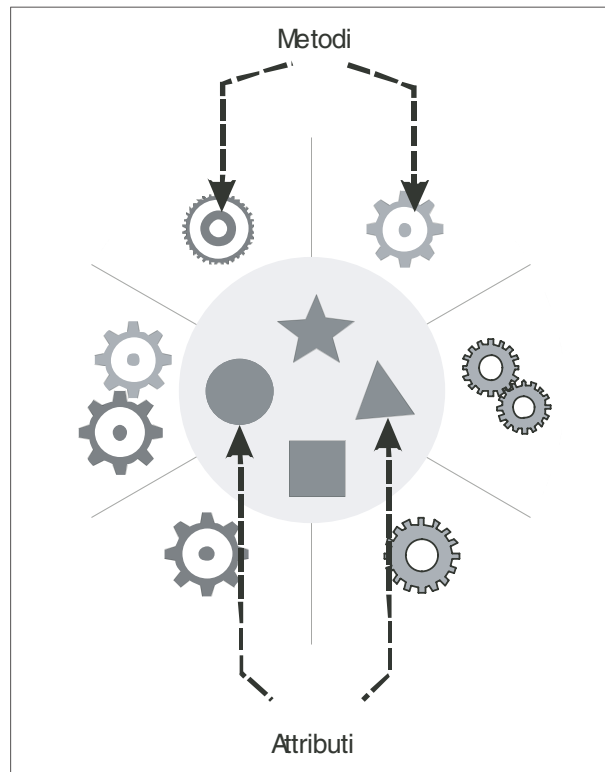
**Figura 5.1 –** *Ogni oggetto può essere descritto mediante il valore dei suoi attributi.*

## Le modalità di utilizzo di un oggetto

Ogni oggetto ha specifiche modalità di utilizzo: il coltello può tagliare, il fornello può scaldare, la forchetta può infilzare. Alcuni oggetti hanno una sola modalità di utilizzo, mentre altri ne hanno più di una. Un normale coltello da cucina, tanto per fare un esempio, ha almeno tre modalità di utilizzo: taglia, spalma e infilza.

Le modalità di utilizzo di un oggetto caratterizzano il suo comportamento, dal momento che esse costituiscono l'unica via per interagire con l'oggetto stesso. Per portare a termine compiti complessi, normalmente si utilizzano più oggetti, facendoli interagire tra loro. Le modalità di utilizzo prevedono di solito limiti espliciti alla possibilità di interazione con altri oggetti: un coltello da cucina permette di sbucciare una mela, ma non di abbattere un albero, cosa che invece è possibile fare con la lama di un machete. Il coltello da cucina e il machete sono oggetti simili tra loro, dal momento che sono entrambi attrezzi da taglio, ma è evidente a livello intuitivo che le rispettive modalità di utilizzo sono diverse.

# La metafora degli oggetti nella programmazione

Un oggetto è un'entità software dotata di uno stato (i suoi attributi) e di un insieme di metodi che permettono all'utente di interagire con esso. Lo stato di un oggetto è accessibile solo tramite i suoi metodi: per questo, spesso gli oggetti vengono rappresentati come "contenitori" di attributi, che mostrano al proprio esterno soltanto i metodi.

**Figura 5.2 –** *Rappresentazione grafica di un oggetto software.*

Gli oggetti software, al pari degli oggetti reali, possiedono una loro coerenza, che ne favorisce il corretto utilizzo. Verrà ora mostrato come si creano e si utilizzano gli oggetti in un qualsiasi programma Java: sarà interessante notare la somiglianza tra un algoritmo che fa uso di oggetti e la descrizione dell'operazione di cottura di una patata vista nei paragrafi precedenti.

## Creazione di un oggetto

Prima di usare un oggetto all'interno di un programma è necessario crearlo. La creazione di un oggetto richiede tre fasi: dichiarazione, creazione e assegnamento, in modo simile a quello che avviene per gli array. Per esempio, ecco le fasi di creazione di un ipotetico oggetto "Patata", a partire dalla dichiarazione:

```
Patata p;
```

La dichiarazione costruisce una variabile dello stesso tipo dell'oggetto che si desidera creare.

Tale variabile non è l'oggetto vero e proprio bensì un reference, ossia una specie di "centrale di controllo" che permette di comunicare con l'oggetto vero e proprio. Senza una centrale di controllo, l'oggetto risulta irraggiungibile dal programma, e gli oggetti irraggiungibili vengono classificati come rifiuti (garbage) ed eliminati. Come si è già visto per i vettori, è possibile avere più di un reference allo stesso oggetto. La creazione e l'assegnamento richiedono l'uso della parola riservata new:

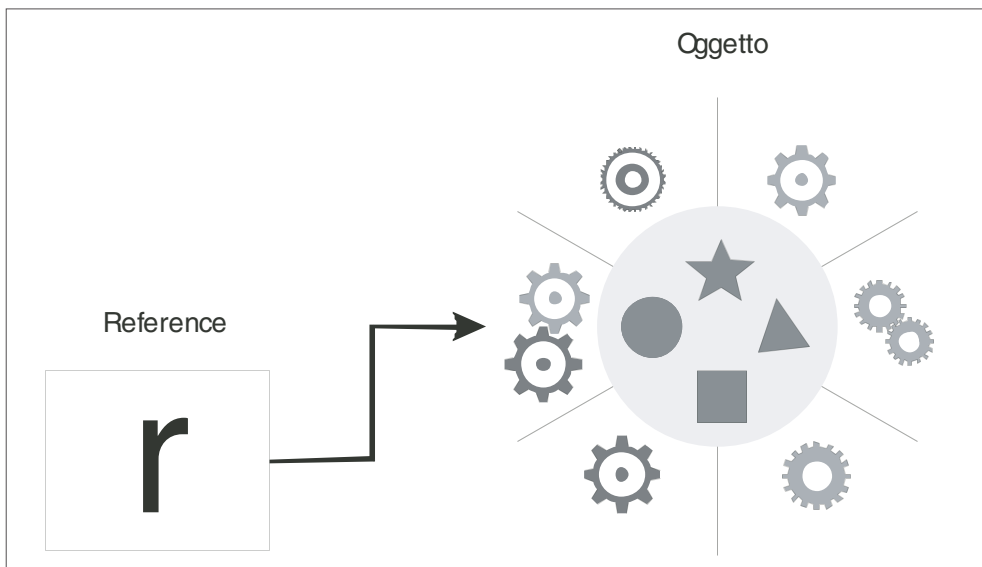
```
p = new Patata();
```

Ovviamente è possibile dichiarare, creare e assegnare un oggetto in un'unica riga:

```
Patata p = new Patata();
```

Alcuni oggetti richiedono uno o più parametri in fase di creazione: tali parametri sono necessari per impostare lo stato iniziale dell'oggetto. Si immagini un oggetto "Cucina" che permetta di stabilire, in fase di creazione, il numero di fornelli a gas e di quelli elettrici: per creare una cucina con quattro fornelli a gas e uno elettrico sarebbe necessario utilizzare un'istruzione del tipo:

```
Cucina c = new Cucina(4,1);  // quattro fornelli a gas e uno elettrico
```

**Figura 5.3 –** *Un reference è una centrale di controllo che permette di inviare le direttive all'oggetto vero e proprio.*

## Il comportamento di un oggetto e i suoi metodi

Per eseguire un'azione su un oggetto è necessario effettuare una chiamata a metodo. Tale operazione richiede tre informazioni: l'oggetto su cui invocare il metodo, il nome del metodo e i parametri richiesti dal metodo stesso.

La sintassi della chiamata a metodo in Java ha la forma:

```
oggetto.metodo(p1,p2,…,pn);
```

I metodi possono anche restituire un valore; il valore di ritorno e i parametri possono essere sia valori primitivi (int, boolean, float ecc.) sia oggetti.

Nell'esempio seguente:

```
Soldi s = banca.riscuoti(assegno);
```

la chiamata al metodo `riscuoti()` dell'oggetto `banca` applicata a un oggetto di tipo `assegno` restituisce un oggetto di tipo `Soldi`.

## Gli attributi di un oggetto e il suo stato

Gli attributi sono valori che descrivono lo stato di un oggetto. In un oggetto software, gli attributi sono accessibili solo tramite i metodi; questa proprietà permette a chi progetta l'oggetto di proteggere l'integrità dell'oggetto stesso in almeno due maniere:

- Impedendo la modifica di un attributo al fine di renderlo accessibile solo in lettura.
- Imponendo precise restrizioni ai valori che un determinato attributo può assumere, in modo da vietare l'uso improprio dell'oggetto.

Di norma, gli attributi sono accessibili mediante opportuni metodi, detti getter e setter", così definiti a causa del prefisso "get" o "set".. I metodi getter permettono di leggere il valore di un attributo, e i setter di modificarlo:

```
int saldo = contoCorrente.getSaldo();        // interroga il conto corrente per conoscere il saldo
automobile.setRapportoDiCambio(4);           // imposta la quarta marcia sull'oggetto automobile
```

In alcuni casi, gli attributi non possono essere modificati direttamente tramite un metodo setter: essi cambiano in seguito alla chiamata di altri metodi che producono una transizione di stato complessa. Nell'esempio della cucina, è abbastanza intuitivo pensare che una pentola disponga di un metodo `getTemperatura()` ma non del corrispondente metodo `setTemperatura()`, dato che l'unico modo per alzare la temperatura di una pentola è quella di scaldarla sul fuoco:

```
fornello.accendi();                       // accende il fornello
while(pentola.getTemperatura() < 100)     // scalda la pentola fino a che non raggiunge i 100°C
   fornello.scalda(pentola);
fornello.spegni();                        // spegne il fornello
```

## Interazione complessa tra oggetti

Per mostrare un esempio concreto di interazione tra oggetti, verrà ora mostrato come la procedura di cottura di una patata, descritta in modo informale nei paragrafi precedenti, verrebbe descritta in Java:

```
Fornello f = new Fornello();              // Crea il Fornello
Coltello c = new Coltello();              // Crea il Coltello
Patata p = new Patata();                  // Crea la Patata
Rubinetto r = new Rubinetto();            // Crea il Rubinetto
Pentola pent = new Pentola();             // Crea la Pentola
Forchetta fork = new Forchetta();         // Crea la Forchetta

While(p.èSporca())                        // Fino a che la patata è sporca
   r.risciaqua(p);                        // Risciacqua la patata con il rubinetto
c.sbuccia(p);                             // Sbuccia la patata con il coltello
pent.riempi(r);                           // Riempi la pentola con il rubinetto
f.accendi()                               // Accendi il fornello
pent.inserisci(p);                        // Metti la patata nella pentola
while(p.èCruda())                         // Finché la patata è cruda
f.scalda(pent);                           // scalda la pentola sul primo fornello

f.spegni();                               // Spegni il primo fornello
fork.inforca(p)                           // Raccogli la patata con la forchetta
```

Si può già notare come gli oggetti aggiungano espressività al codice, e come tendano a fornire una dimensione materiale alle entità evanescenti che compongono un programma. Come diverrà più chiaro in seguito, il ricorso agli oggetti fornisce un approccio concettuale di altissimo livello a qualsiasi contesto applicativo, favorendo la modularità e la riutilizzabilità dei diversi componenti di un programma. Grazie agli oggetti, il lavoro di programmazione diventa qualcosa di simile a un gioco di costruzioni a incastro, come il Lego.

# Oggetti di sistema

Dopo aver introdotto in modo informale l'uso degli oggetti nella programmazione, è ora di studiare alcuni oggetti di sistema che vengono usati nella grande maggioranza dei programmi realizzati con Java. Il primo di questi è la stringa, che permette di manipolare sequenze di caratteri e comporre parole e frasi. A differenza che in altri linguaggi, come C e Pascal, in Java le

stringhe vengono implementate come oggetti, con il vantaggio di permetterne un uso ad alto livello. Dopo le stringhe verranno illustrati i vettori dinamici e le mappe hash, due importanti strutture di dati. Infine, verranno trattate le wrapper class, che permettono di trattare come oggetti anche i valori appartenenti ai tipi primitivi.

# Stringhe

L'oggetto String contiene una sequenza di caratteri di lunghezza arbitraria, che può essere utilizzata per memorizzare parole, frasi o testi di qualsiasi dimensione. Il contenuto di un oggetto String viene deciso al momento della sua creazione, e non può essere modificato in seguito:

```
String s = new String("Questo è un esempio di stringa di testo");
```

I metodi di String permettono di ispezionare il contenuto della stringa, di estrarne dei frammenti (sottostringhe), di verificare l'uguaglianza con un'altra stringa e di effettuare altre interessanti operazioni. Nei prossimi paragrafi verranno analizzate a fondo le caratteristiche di questo oggetto fondamentale.

## Le stringhe e Java: creazione, concatenazione e uguaglianza

Le stringhe sono uno strumento così importante nella programmazione che i progettisti di Java hanno deciso di introdurre dei costrutti nel linguaggio per semplificarne l'uso.

### Creazione

In fase di creazione, invece di ricorrere all'operatore new, è possibile utilizzare direttamente un letterale stringa, ossia una qualsiasi sequenza di caratteri racchiusa tra doppi apici come nel seguente esempio:

```
String s = "Ciao";                    // è eqivalente a String s = new String("Ciao");
```

Ogni letterale stringa viene considerato automaticamente come un oggetto String, pertanto è possibile applicare la chiamata a metodo direttamente su di esso. La seguente riga di codice:

```
System.out.println("Ciao".length());
```

stampa a schermo la *lunghezza* della stringa, ossia il risultato della chiamata del metodo length() sulla stringa "Ciao", e non il suo contenuto.

## Concatenazione

Java prevede l'uso di + come operatore di concatenazione in alternativa al metodo concat(String s). Grazie all'operatore +, un'istruzione come:

```
String s = "Ecco ".concat("una ").concat("stringa ").concat("concatenata");
```

può essere sostituita dalla più sintetica e intuitiva:

```
String s = "Ecco " + "una " + "stringa " + " concatenata";
```

L'operatore di concatenazione è particolarmente utile quando si devono creare stringhe così lunghe da occupare più di una riga. Dal momento che i letterali stringa non possono essere interrotti con un ritorno a capo, è possibile concatenare più letterali stringa da una riga ciascuno ricorrendo all'operatore +:

```
String s = "La vispa Teresa, " +
           "avea tra l'erbetta " +
            "a volo sorpresa " +
            "gentil farfalletta";
```

L'operatore + effettua in modo automatico e trasparente la conversione dei tipi primitivi in stringa:

```
int risultato = 12 * a;
System.out.println("Il contenuto della variabile risultato è: " + risultato);
```

## Operatore di uguaglianza e metodo equals

Quando si lavora su oggetti o vettori, bisogna fare molta attenzione all'uso dell'operatore di uguaglianza == che, a differenza di quanto ci si potrebbe aspettare, non serve a testare l'uguaglianza di contenuto, ma solo l'*uguaglianza di riferimento*. In figura 5.4 si vedono due variabili s1 e s2 che puntano a due distinti oggetti stringa. Nonostante entrambe le stringhe contengano la parola "Ciao!", il test con l'operatore == restituirà false, dal momento che gli oggetti puntati dalle due variabili sono due oggetti differenti, localizzati in due zone di memoria distinte. Per verificare se due stringhe hanno lo stesso contenuto, come in questo caso, è necessario ricorrere al metodo equals(String s).

In figura 5.5, invece, le variabili s1 e s2 puntano allo stesso oggetto stringa: in questo caso, il test con l'operatore == restituirà true. Scenari di questo tipo si presentano ogni volta che si assegna lo stesso oggetto a più di una variabile:

```
s1 = "Ciao! ";
s2 = s1;                                    // s1 e s2 puntano allo stesso oggetto stringa
```

**Figura 5.4 –** *Le variabili s1 e s2 puntano a due oggetti stringa con lo stesso contenuto.*

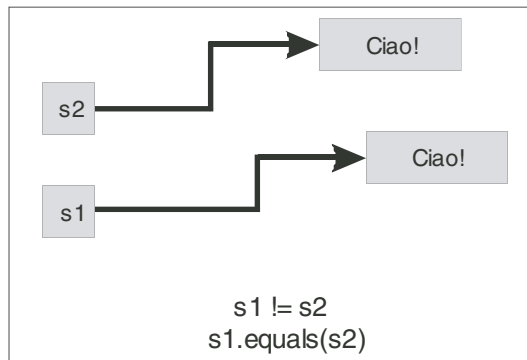
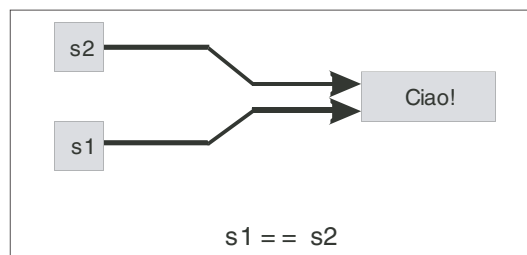



**Figura 5.5 –** *Le variabili s1 e s2 puntano allo stesso oggetto stringa.*

## Operazioni fondamentali

L'oggetto `String` comprende più di 50 metodi; quelli fondamentali, comunque, sono appena 6:

- `String substring(int beginIndex , int endIndex )`: restituisce una sottostringa che contiene il testo incluso tra `beginIndex` e `endIndex`-1.

- `char charAt(int i)`: restituisce l'i-esimo carattere della stringa.

- `int length()`: restituisce la lunghezza della stringa.

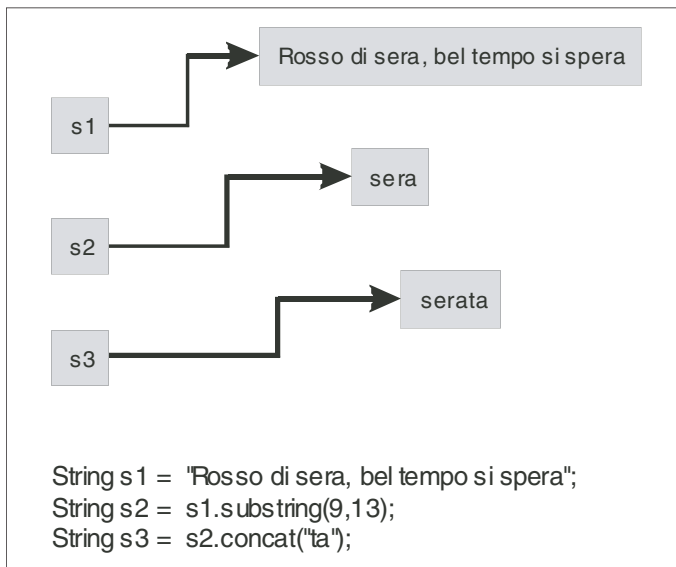
- `int compareTo(String anotherString)`: effettua una comparazione con un'altra stringa, e restituisce: 0 se l'argomento è una stringa uguale; un valore inferiore a 0 se l'argomento è una stringa più avanti rispetto all'ordine alfabetico; un valore superiore a 0 in caso contrario. Se si desidera che nella comparazione non venga considerata la differenza tra maiuscole e minuscole, si può utilizzare il metodo `compareToIgnoreCase(String s)`.

- String concat(String str): effettua la concatenazione con la stringa str. Come si è già visto, è equivalente all'operatore +.

- boolean equals(Object o): restituisce true se la stringa risulta uguale all'oggetto passato come parametro. Se si desidera che venga ignorato l'ordine tra maiuscole e minuscole, si può utilizzare il metodo equalsIgnoreCase(String s).

Si noti che i metodi concat() e substring() non modificano l'oggetto stringa su cui vengono invocati, ma creano un nuovo oggetto che ha per valore il risultato dell'operazione. In figura 5.8 si può osservare una rappresentazione della memoria dopo l'esecuzione delle seguenti istruzioni:

```
String s1 = "Rosso di sera, bel tempo si spera";
String s2 = s1.substring(9,13);
String s3 = s2.concat("ta");
```

**Figura 5.6 –** *Rappresentazione della memoria del computer dopo l'esecuzione di una serie di operazioni su stringhe.*

## Altri metodi utili

L'oggetto String prevede diversi altri metodi utili. Ecco un gruppo di metodi che effettuano interrogazioni:

- boolean startsWith(String prefix): restituisce true se la stringa inizia con il prefisso specificato dal parametro.

- boolean endsWith(String suffix): restituisce true se la stringa finisce con il suffisso specificato dal parametro.

- int indexOf(String str): scandisce la stringa dall'inizio alla fine, e verifica se contiene o meno la stringa specificata come parametro. Se la trova, restituisce l'indice del primo carattere della sotto stringa cercata, altrimenti restituisce –1.

Un esempio di uso di questi metodi è il controllo di validità dell'estensione di un file:

```
import javax.swing.*;

public class ProvaEndsWith {

  public static void main(String argv[]) {
    String file = JOptionPane.showInputDialog(null, "Inserisci il nome di un file java valido");
    if( file.endsWith(".java") )
      JOptionPane.showMessageDialog(null,"Il nome del file è valido");
    else
      JOptionPane.showMessageDialog(null,"Il nome del file non termina con .java");
  }
}
```

Un altro gruppo di metodi fornisce alcune funzionalità utili in fase di manipolazione:

- String replace(char oldChar , char newChar): restituisce una stringa in cui il carattere specificato dal primo parametro è stato sostituito da quello indicato dal secondo.

- String toLowerCase(): restitituisce una stringa in cui tutti i caratteri sono minuscoli.

- String toUpperCase(): restitituisce una stringa in cui tutti i caratteri sono maiuscoli.

- String trim(): restituisce una stringa dalla quale sono stati rimossi gli eventuali spazi all'inizio e alla fine.

Come esempio riepilogativo, ecco un programma che inverte le frasi fornite in input:

```
import javax.swing.*;

public class Invertitore {

  public static void main(String argv[]) {
```

```
      String s = JOptionPane.showInputDialog(null, “Inserisci una frase”);
      s = s.trim();
      String inversa = “”;

      for( int i = s.length() -1 ; i >= 0  ; i-- )
        inversa = inversa + s.charAt(i);

      JOptionPane.showMessageDialog(null,inversa);
    }
  }
```

Il programma chiede all’utente di inserire una frase, quindi la scandisce lettera per lettera dall’ultima alla prima e le aggiunge alla stringa “inversa”, creando in tal modo una frase speculare rispetto a quella inserita dall’operatore.

in questi e nei prossimi esempi si farà uso di piccole finestre di dialogo per l’input e l’output dei dati sullo schermo. La forma base dell’istruzione di input è:

String s = JOptionPane.showInputDialog(null, *messaggio*)

Questa istruzione mostra all’utente una finestra di dialogo con il messaggio specificato dal parametro, e restituisce la stringa inserita dall’operatore. Per creare invece una finestra di output che visualizzi a schermo un messaggio si ricorre alla seguente istruzione:

JOptionPane.showMessageDialog(null,*messaggio*);

Se si desidera utilizzare queste istruzioni nei propri programmi, è necessario inserire la direttiva:

```
import javax.swing.*;
```

all’inizio del programma. Una descrizione completa dei controlli grafici e del loro utilizzo è rimandata ai capitoli 12, 13, 14 e 15.

# Vettori dinamici

Nel capitolo 2 sono stati introdotti gli array, che permettono di lavorare su insiemi di variabili di lunghezza prefissata. Il principale difetto degli array è la loro natura statica: la dimensione viene stabilita al momento della creazione, e non può cambiare in alcun modo. Questa limitazione viene superata dai vettori dinamici, entità software simili agli array ma senza il limite della dimensione prefissata. Le librerie Java offrono un buon numero di vettori dinamici, simili tra

loro nell'interfaccia di programmazione ma diversi nell'implementazione interna: il più usato di questi è senza dubbio Vector.

## Uso di Vector

Vector è un oggetto concettualmente simile a un array, ma a differenza di quest'ultimo prevede un utilizzo esclusivamente orientato agli oggetti. Esso va in primo luogo creato allo stesso modo di qualsiasi altro oggetto Java:

```
Vector v = new Vector();
```

Successivamente, esso può essere riempito mediante il metodo add(Object o), che aggiunge in coda alla lista l'oggetto passato come parametro:

```
String nome = "Un nome";
v.add(nome);
```

A differenza dell'array, che viene creato in modo da contenere valori di un certo tipo, il Vector memorizza oggetti di tipo generico: per questa ragione, al momento del prelievo, è necessario ricorrere all'operatore di casting per riconvertire l'elemento al suo tipo di origine:

```
String n = (String)v.get(1);
```

L'approccio completamente orientato agli oggetti ha permesso di incorporare un certo numero di funzionalità all'interno di Vector, e di renderle disponibili sotto forma di metodi. Per esempio, se si desidera conoscere la posizione di un elemento nella lista, è sufficiente chiamare l'apposito metodo di ricerca:

```
int position = v.indexOf("Un nome");
```

Se invece si vuole creare una copia del vettore, si può utilizzare il metodo clone():

```
Vector nuovoVector = (Vector)v.clone();
```

> Nota: per usare il Vector all'interno di un programma è necessario aggiungere la direttiva
>
> ```
> import java.util.*;
> ```
>
> in testa al programma. Il significato e l'uso della direttiva import verranno chiariti nel capitolo 11.

## Metodi fondamentali di Vector

Ecco un elenco e una descrizione dei 9 metodi principali di Vector:

- boolean add(Object o): aggiunge l'elemento in coda alla lista.
- void add(int index , Object element): aggiunge un elemento nella posizione specificata; gli elementi successivi vengono spostati in avanti di uno.
- void clear(): svuota completamente la lista.
- boolean isEmpty(): restituisce true se la lista è vuota.
- Object get(int i): restituisce l'i-esimo elemento della lista.
- int indexOf(Object o): restituisce l'indice dell'elemento passato come parametro, o –1 se l'elemento non è presente nella lista.
- Object remove(index i): rimuove l'i-esimo elemento della lista, e sposta l'indice di tutti gli elementi successivi in avanti di un valore. Il metodo restituisce l'elemento appena rimosso.
- Object set(int i , Object element): mette l'elemento specificato in i-esima posizione, in sostituzione dell'elemento preesistente. Il metodo restituisce l'elemento appena rimosso.
- int size(): restituisce la dimensione della lista.

## Iterator

Per effettuare una determinata operazione su tutti gli elementi di un vettore dinamico, è possibile ricorrere a un ciclo for simile a quelli che si usano con gli array:

```
for(int i=0;i<v.size();i++)
  System.out.println((String)v.get(i));
```

Tuttavia, per semplificare questo tipo di operazione e per aggiungere un tocco di eleganza, è possibile ricorrere a una soluzione orientata agli oggetti che prevede il ricorso a un apposito oggetto, Iterator, che può essere prelevato dal Vector mediante un apposito metodo:

```
Iterator i = nomi.iterator();
```

Iterator è un oggetto molto semplice e intuitivo, che permette di scandire la lista dal primo all'ultimo elemento per effettuare una determinata operazione su ciascuno dei suoi elementi. L'utilizzo canonico di un Iterator ha la forma:

```
while(i.hasNext()) {                          // verifica se c'è un ulteriore elemento
  String s = (String)i.next();                // preleva l'elemento
  System.out.println(s);                      // lo utilizza
}
```

## Conversione in array

Il Vector dispone anche di un metodo che permette di trasferire il suo contenuto in un array:

```
Object[] toArray(Object[] a)
```

L'array fornito come parametro viene ridimensionato, riempito con gli elementi del Vector e restituito all'utente. Dal momento che il Vector memorizza gli elementi con il tipo generico Object, è necessario passare come parametro un array del tipo giusto, e utilizzare l'operatore di casting sul valore di ritorno:

```
String[] lista = (String[])v.toArray(new String[0]);
```

## Un esempio riepilogativo

Un piccolo esempio permetterà di illustrare un uso tipico di Vector. All'utente viene richiesto di inserire un elenco di nomi mediante una serie di finestre di dialogo; non appena l'utente avrà segnalato, premendo il pulsante Annulla, la conclusione della fase di inserimento, il vettore verrà scandito con un Iterator, e i suoi valori verranno stampati a schermo:

```
import java.util.*;
import javax.swing.*;

public class ListaNomi {

  public static void main(String argv[]) {
    Vector v = new Vector();

    while(true) {
      String nome = JOptionPane.showInputDialog(null, "Inserisci un nome");
      if(nome == null || nome.equals(""))
        break;
      else
        v.add(nome);
    }

    Iterator i = v.iterator();

    System.out.println("I nomi inseriti sono:");
    while(i.hasNext()) {
      System.out.println((String)i.next());
    }
  }
}
```

# Mappe hash

La mappa hash è un contenitore di oggetti simile al vettore, che memorizza una collezione di oggetti associandoli a stringhe di testo invece che alla loro posizione. La mappa hash più comunemente utilizzata è Hashtable:

```
Hashtable h = new Hashtable();
```

Per depositare elementi nella mappa, bisogna utilizzare il metodo put(Object chiave,Object valore), che richiede come chiave un valore univoco (tipicamente una stringa di testo, ma può andare bene qualsiasi tipo di oggetto) e come valore l'oggetto da memorizzare:

```
h.put("Nome","Mario");
h.put("Cognome","Rossi");
h.put("Età","25 anni");
h.put("Lavoro","Insegnante");
```

**Tabella 5.1 –** *Una tabella hash è una collezione di oggetti, in cui gli elementi sono indicizzati tramite stringhe chiave.*

| Chiave | Valore |
|---|---|
| Nome | Mario |
| Cognome | Rossi |
| Età | 25 anni |
| Lavoro | Insegnante |

La mappa può ovviamente contenere elementi duplicati; non può invece associare più di un elemento alla stessa chiave. Per recuperare l'elemento, sarà sufficiente fornire la chiave attraverso il metodo get:

```
String s = (String)h.get("nome");
```

Anche in questo caso, in fase di recupero è necessario ricorrere all'operatore di casting.

## Metodi principali

I metodi principali delle mappe hash sono:

- Object put(Object key , Object value): inserisce un nuovo oggetto nella mappa, associandolo alla chiave specificata.

- Object get(Object key): preleva dalla mappa l'oggetto associato con la chiave specificata.
- Object remove(Object key): rimuove dalla mappa l'oggetto associato alla chiave specificata.
- void clear(): svuota la mappa.
- int size(): restituisce il numero di coppie chiave-valore contenute nella mappa.
- boolean isEmpty(): restituisce true se la mappa è vuota.
- boolean containsKey(Object key): verifica se la mappa contiene la chiave specificata.
- boolean containsValue(Object value): verifica se la mappa contiene il valore specificato.

## Estrazione dell'insieme di chiavi o valori

Per ottenere l'insieme delle chiavi o dei valori, esiste un'apposita coppia di metodi:

- Set keySet(): restituisce l'insieme delle chiavi.
- Collection values(): restituisce la lista dei valori.

Questi metodi restituiscono oggetti di tipo Set e Collection. Entrambi questi oggetti hanno metodi comuni a Vector, e in particolare il metodo iterator() che rende possibile la scansione degli elementi:

```
Iterator keyIterator = h.keySet().iterator();
System.out.println("L'insieme delle chiavi è:");
while(keyIterator.hasNext())
  System.out.println((String)keyIterator.next());

Iterator elementIterator = h.values().iterator();
System.out.println("L'insieme degli elementi è:");
while(elementIterator.hasNext())
   System.out.println((String)elementIterator.next());
```

# Wrapper class

Vector e Hashtable possono memorizzare solamente oggetti: se si desidera memorizzare al loro interno valori primitivi è necessario racchiuderli in apposite wrapper class (classi involucro). Il linguaggio Java prevede una wrapper class per ogni tipo primitivo: Boolean, Byte, Short, Integer, Long, Float e Double. Il loro uso è abbastanza semplice: in fase di creazione è sufficiente specificare nel costruttore il valore da inglobare:

```
Integer i = new Integer(15);
```

È disponibile anche un costruttore che accetta valori sotto forma di String, una funzionalità estremamente utile in fase di inserimento dati:

```
String numeroStringa = JOptionPane.showInputDialog(null, "Inserisci un numero intero");
Integer numeroInteger = new Integer(numeroStringa);
int numero = numeroInteger.intValue();
```

Per recuperare il valore nel suo formato naturale, ogni wrapper class dispone di un apposito metodo:

| | |
|---|---|
| boolean BooleanValue() | su oggetti di tipo Boolean. |
| byte byteValue() | su oggetti di tipo Byte. |
| short shortValue() | su oggetti di tipo Short. |
| int intValue() | su oggetti di tipo Integre. |
| long longValue() | su oggetti di tipo Long. |
| float floatValue() | su oggetti di tipo Float. |
| double doubleValue() | su oggetti di tipo Double. |

# Le classi in Java

Andrea Gini

Dopo aver introdotto in modo informale l'uso degli oggetti in Java, è giunto il momento di discutere in modo esteso e formale tutti gli aspetti della programmazione a oggetti. La classe è un costrutto che permette di raggruppare, in un pacchetto indivisibile, un gruppo di variabili e un insieme di metodi che hanno accesso esclusivo a tali variabili:

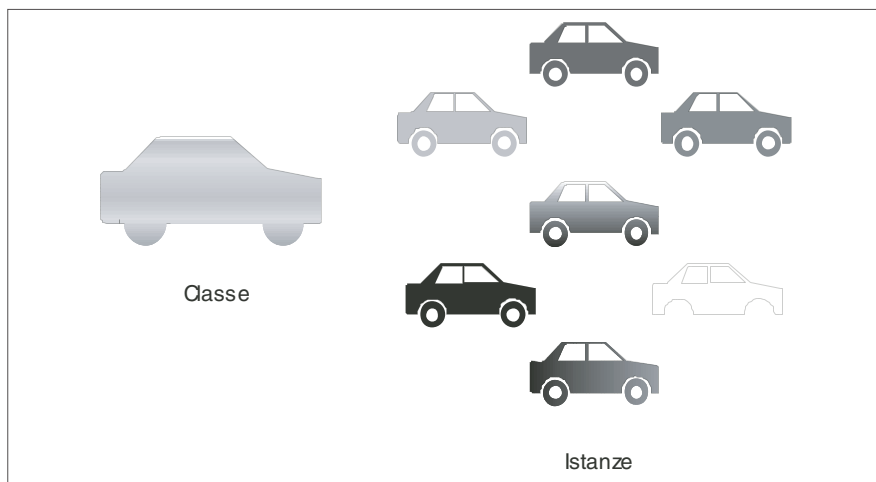
```
public class nome {
  private tipo attributo1;
  private tipo attributo2;
  ….

  public tipo metodo1() {
    // corpo del metodo
  }
  public tipo metodo2(tipo parametro1 , tipo parametro2) {
    // corpo del metodo
  }
}
```

Gli attributi sono variabili accessibili da qualsiasi metodo interno alla classe, ma inaccessibili dall'esterno. I metodi sono procedure che possono operare sia sui dati passati come parametri, sia sugli attributi della classe.

Prima di proseguire oltre, è necessario chiarire la differenza concettuale tra classe, istanza e reference. La classe è la matrice sulla quale vengono prodotti gli oggetti. Essa è come uno stampo, che permette di plasmare un materiale informe per produrre una molteplicità di oggetti

simili tra loro. Per distinguere la matrice dal prodotto, il lessico della programmazione orientata agli oggetti ricorre a due termini: classe e istanza.

La classe, corrispondente al codice sorgente scritto dal programmatore, è presente in singola copia nella memoria del computer. Ogni volta che si ricorre all'operatore `new`, viene creata una nuova istanza della classe, ossia un oggetto di memoria conforme alle specifiche della classe stessa.

**Figura 6.1 –** *La classe è come uno stampo, capace di generare un'infinità di oggetti simili tra di loro ma dotati nel contempo di attributi univoci.*

La variabile a cui viene associato l'oggetto, d'altra parte, è soltanto un reference: essa è simile a un telecomando con il quale è possibile inviare delle direttive all'oggetto vero e proprio. Ogni volta che si invoca un metodo su una variabile, la variabile in sé non subisce nessun cambiamento: la chiamata a metodo viene inoltrata all'oggetto vero e proprio, che reagisce all'evento nelle modalità previste dal codice presente nella classe. Questa architettura permette di avere più variabili che puntano allo stesso oggetto: l'oggetto è unico, indipendentemente dal numero di reference usati per inoltrare le chiamate a metodo. Il reference può anche non puntare alcun oggetto; in questo caso, esso assume il valore speciale `null`. Al momento della dichiarazione, se non viene specificato diversamente, ogni reference ha valore `null`.

# Incapsulamento

L'incapsulamento è un principio di progettazione che prevede che il contenuto informativo di una classe rimanga nascosto all'utente, in modo tale che i metodi siano l'unica via per interagire con un determinato oggetto. Questo approccio presenta almeno due grossi vantaggi: innanzitutto,

esso permette al programmatore di disciplinare l'accesso agli attributi di una classe, in modo da impedire che ne venga fatto un uso sbagliato; in secondo luogo, l'incapsulamento consente a chi utilizza una classe di concentrarsi esclusivamente sull'interfaccia di programmazione, tralasciando ogni aspetto legato all'implementazione.

Quasi tutti gli oggetti del mondo reale presentano questa stessa proprietà. Si pensi a un telefono cellulare: per capirne il funzionamento interno è necessaria una laurea in ingegneria elettronica, mentre bastano pochi minuti per imparare a comporre un numero e mettersi in contatto con qualcuno. Questa filosofia di progettazione ha due importanti implicazioni: per prima cosa, imparare a *usare* un oggetto è di norma più facile che capire come funziona; inoltre, una volta appreso il funzionamento comune a una classe di oggetti (automobili, telefoni, forbici ecc.), diventa facile utilizzare qualsiasi altro oggetto di quella particolare classe (una Fiat Punto, una Renault Clio e così via).

# Dichiarazione di metodo

La struttura standard di un metodo senza parametri è la seguente:

```
public tipo nome() {
  istruzione1;
  istruzione2;
  ....
  instruzioneN;

  return returnVal;
}
```

La sequenza `public` *`tipo nome`*`()` viene detta *firma* del metodo (in inglese, *signature*). Come tipo è possibile specificare uno qualsiasi dei tipi primitivi visti in precedenza (`int`, `long`, `short`, `byte`, `char`, `boolean`, `float` e `double`) o il tipo di un oggetto. Esiste anche il tipo speciale `void`, da usare quando il metodo non restituisce nessun valore. Tutti i metodi con tipo diverso da `void` devono terminare con l'istruzione `return`, seguita da un valore o da una variabile dello stesso tipo presente nella firma del metodo.

```
public class Mela {
    private int larghezza;
    private int lunghezza;
    private Color colore;

    public int getLarghezza() {
      return larghezza;                    // restituisce un intero
    }
    public int getLunghezza() {
      return lunghezza;                    // restituisce un intero
    }
    public Color getColore() {
```

```
    return colore;                              // restituisce un oggetto di tipo Color
  }
}
```

## Dichiarazione di metodo con parametri

L'uso dei metodi può essere potenziato notevolmente grazie ai parametri. I parametri permettono di generalizzare i metodi, in modo da estenderne l'uso a diversi contesti a seconda del valore dei parametri stessi. Per dichiarare un metodo parametrizzato, è necessario seguire una sintassi un po' diversa da quella vista in precedenza: dopo il nome del metodo, tra una coppia di parentesi tonde, bisogna specificare una serie di coppie tipo-nome, in maniera simile a come si fa con le dichiarazioni di variabile:

```
public tipo nome(tipo1 nome1,tipo2 nome2, .... , tipoN nomeN) {
  istruzione1;
  istruzione2;
  ....
  instruzioneN;
}
```

All'interno del metodo i parametri possono essere trattati come normali variabili. Nell'esempio seguente viene definito un metodo min, che prende in input una coppia di interi e restituisce il minore tra i due:

```
public int min(int n1 , int n2) {
  if ( n1 < n2 )
    return n1
  else
    return n2;
}
```

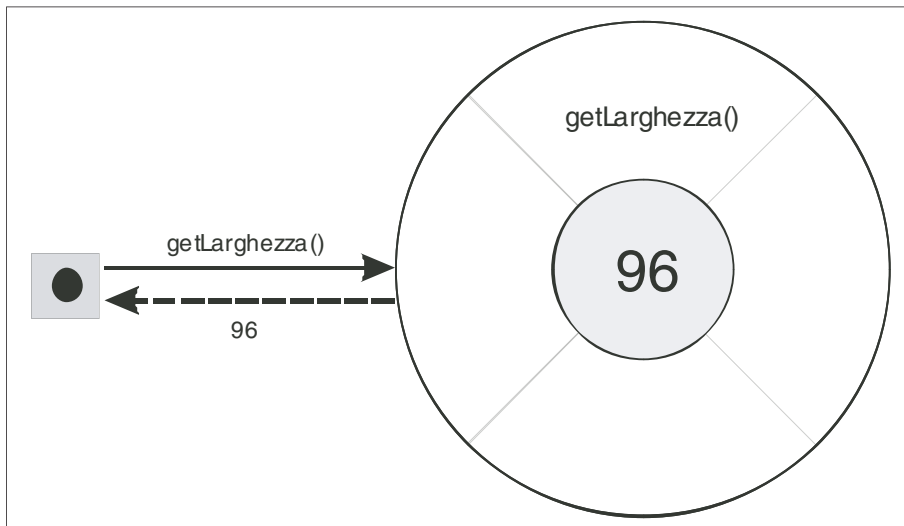
# Chiamata a metodo: la dot notation

Per invocare un metodo su un oggetto, è necessario applicare l'operatore . su un reference che punti all'oggetto, specificando il nome del metodo da chiamare e gli eventuali parametri:

```
Mela m = new Mela();
Int larghezza = m.getLarghezza();               // Chiamata a metodo
```

Se il metodo invocato fa parte della classe chiamante, la chiamata non deve essere preceduta da alcun identificatore.

La chiamata a metodo denota un protocollo a scambio di messaggi: chi detiene il reference invia un messaggio di chiamata a metodo, specificando il nome del metodo da invocare e l'elenco dei parametri; l'oggetto chiamato, come risposta, restituisce un valore di ritorno.

**Figura 6.2 –** *Scambio di messaggi durante una chiamata a metodo.*

L'operatore . può essere usato anche per accedere agli attributi public di un oggetto:

```
public class Point {                    // classe con attributi pubblici
  public int x;
  public int y;
}
Point p = new Point();
p.x = 10;                               // assegnamento sull'attributo x della classe Point
```

La notazione che fa uso dell'operatore . prende comunemente il nome di dot notation, o notazione puntata. Nei prossimi paragrafi verranno illustrate le regole di utilizzo della dot notation per accedere alle classi contenute all'interno di package, e a metodi e ad attributi statici. È possibile applicare l'operatore . a cascata sui valori restituiti da un metodo o da un attributo, ottenendo in tal modo l'inoltro della chiamata all'oggetto restituito dalla chiamata precedente. Per esempio, la chiamata:

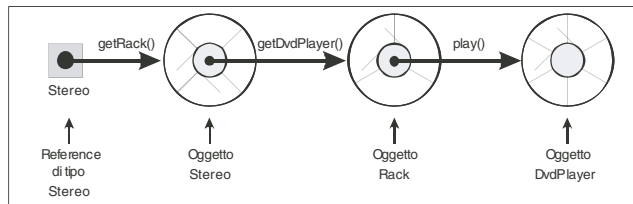
```
// chiama il metodo darker() sul valore restituito dal
// metodo getColor() dell'oggetto riferito dalla variabile m
Color c = m.getColor().darker();
```

è equivalente alla seguente serie di istruzioni:

```
Color coloreMela = m.getColor();
Color c = coloreMela.darker();
```

Simili catene possono avere una lunghezza indefinita, e permettono di operare su qualsiasi oggetto accessibile per via diretta o indiretta a partire da un reference. Per esempio, per far partire un ipotetico lettore DVD contenuto nel rack di un impianto stereo, si utilizzerà una catena di chiamate di questo tipo:

```
stereo.getRack().getDvdPlayer().play();
```

**Figura 6.3 –** *Un esempio di chiamata di metodo a cascata.*

## Parametro attuale e parametro formale

L'uso di metodi con parametri introduce una problematica concettuale piuttosto sottile: cosa succede quando, nel corso di una chiamata a metodo, si passa come parametro una variabile? Si osservi il seguente programma e si cerchi di immaginare come procede l'esecuzione a partire dal metodo m1():

```
public class ProvaParametri {

  public void m1() {
    int a = 5;
    m2(a);  // chiama m2 usando la variabile a come parametro
    System.out.println(a);
  }
  public void m2(int i) {
    i = 10;
  }
}
```

Cosa succede alla variabile a nel momento in cui viene passata come parametro al metodo m2()? Per poter chiarire come vengano trattati casi come questo è necessario illustrare la differenza tra parametro formale e parametro attuale. I parametri definiti nella firma di un metodo vengono detti *parametri formali*. Tali parametri, all'interno del metodo, vengono trattati come delle variabili, la cui visibilità termina alla fine del metodo stesso. La variabile che viene passata come parametro di un metodo viene detta invece *parametro attuale*: esse hanno significato nel contesto del metodo chiamante, e conservano inalterato il loro valore nonostante la chiamata.

Come verrà spiegato nel prossimo paragrafo, nel caso di passaggio di vettori o di oggetti si verifica un fenomeno che può sembrare un'eccezione alla regola secondo cui il parametro attuale non possa essere modificato dalle istruzioni di un metodo. Si raccomanda di leggere con attenzione il paragrafo al fine di capire questa importante distinzione.

```
public class Esempio {

  public void m1() {
    int p1 = 10;
    float p2 = 10.5F;
    Boolean p3 = true;
    m2(p1,p2,p3);                               // p1, p2 e p3 sono parametri attuali
  }
  public void m2(int i, float f ,  boolean b ) {  // i, f e b sono parametri formali
     ….        // corpo del metodo
  }
}
```

Il compilatore richiede che le variabili fornite come parametri attuali siano dello stesso tipo di quelle richieste dai parametri formali presenti nella firma del metodo, nel rispetto delle regole di casting. Pertanto, se un metodo richiede un parametro long, non verranno segnalati errori se si effettua una chiamata con un int. Nel caso opposto (parametro formale int, parametro attuale long) il compilatore segnalerà un errore, che potrà essere evitato solo ricorrendo al casting.

## Passaggio di parametri by value e by ref

All'atto di chiamare un metodo, l'interprete Java copia il valore dei parametri attuali nelle variabili corrispondenti ai rispettivi parametri formali. Tale comportamento viene detto "passaggio di parametri by value", o "per valore". Mediante il passaggio di parametri by value, si può essere sicuri che qualsiasi modifica ai parametri formali non andrà ad alterare il contenuto delle variabili usate come parametri attuali. Si provi a riconsiderare il programma visto nel paragrafo precedente:

```
public class ProvaParametri {

  public void m1() {
    int a = 5;
    m2(a);  // chiama m2 usando la variabile a come parametro
    System.out.println(a);
  }
  public void m2(int i) {
    i = 10;
  }
}
```

Ora è possibile affermare con sicurezza che la variabile a, utilizzata come parametro attuale nella chiamata a m2(), non verrà alterata in alcun modo dalle istruzioni presenti all'interno del metodo m2() stesso.

Cosa succede invece quando, come parametro, viene utilizzato un array o un altro oggetto invece di un tipo primitivo? Si provi a osservare il seguente esempio:

```
public class Parametri2 {
  public void m1() {
    int[] i = new int[10];
    i[0] = 5;
     m2(i);
     System.out.println(i[0]);
  }
  public void m2(nt[] i) {                    // Il parametro è un oggetto
     i[0] = 10;                                // Il metodo modifica l'oggetto
  }
}
```

Diversamente dal programma precedente, questo esempio restituisce in output il valore 10, quello assegnato al primo elemento dell'array all'interno del metodo. Cosa è successo? Al contrario di quanto avviene con le variabili dei tipi primitivi (int, long ecc.), quando si passa a un metodo un array (o qualsiasi altro oggetto), esso viene passato "by ref", o "per riferimento": in altre parole, il parametro formale i presente nel metodo non punta a una copia dell'array definito nel metodo main, ma punta allo stesso identico array, e qualsiasi modifica effettuata su di esso all'interno del metodo andrà ad agire sull'array stesso. Si noti che questo comportamento non è in contraddizione con la regola descritta nel paragrafo precedente, secondo la quale il parametro attuale non può essere modificato dalle istruzioni presenti all'interno di un metodo. Il parametro attuale, che in questo caso è un reference, non può essere modificato in alcun modo dal codice del metodo: a cambiare in questi casi è l'oggetto puntato dal reference, non il reference stesso.

Alcuni linguaggi, come C, C++, Pascal e Visual Basic, permettono di specificare per ogni singolo parametro se si desidera che venga passato per riferimento o per valore. Questa possibilità, sebbene utile in teoria, finisce inevitabilmente per provocare confusione. Per questo, durante la progettazione di Java si è deciso di imporre per default il passaggio by value per tutti i tipi primitivi e il passaggio by ref di array e oggetti.

# Visibilità delle variabili: variabili locali

Durante la realizzazione di un metodo occorre fare grande attenzione alla visibilità delle variabili, ossia quello che nei testi in lingua inglese viene normalmente definito *scope*. Quando si dichiara una variabile all'interno di un metodo, essa risulta visibile solamente all'interno del metodo stesso: per questo, le variabili dichiarate all'interno di un metodo prendono il nome di *variabili locali*. Ecco un esempio:

```
public class Esempio1 {

  public void metodo1() {
    int a = 10;
    System.out.println(a);
  }
  public void metodo2() {
    a = a++; //ERRORE!
    System.out.println(a);
  }
}
```

Il primo dei due metodi è corretto: esso dichiara la variabile intera a, la inizializza a 10 e la stampa a schermo. Nel secondo metodo c'è invece un errore: si cerca di far riferimento alla variabile a che in questo contesto non è stata definita. Se si corregge il secondo metodo nel modo seguente:

```
  static void metodo2() {
    int a = 10;
    a++;
    System.out.println(a);
  }
```

si ottiene un metodo corretto, che dichiara una nuova variabile a (diversa, si faccia attenzione, dalla omonima variabile a presente nel metodo1), la inizializza a 10, la incrementa a 11 e la stampa.

# Ricorsione

Il codice di un metodo può contenere chiamate a qualsiasi altro metodo: cosa succede nel caso limite in cui un metodo contiene una chiamata a sé stesso? In casi come questo si ottiene un'esecuzione ricorsiva. Per capire l'uso e l'utilità della ricorsione, verrà illustrato un esempio divenuto ormai classico: la funzione fattoriale. La funzione fattoriale viene tipicamente definita per ricorsione, dicendo che il fattoriale di 0 è 1 e che il fattoriale di un qualsiasi numero n intero è pari a n moltiplicato per il fattoriale del numero n - 1. Pertanto il fattoriale di 0 è 1, il fattoriale di 3 è 6 (3*2*1), il fattoriale di 4 è 24 (1*2*3*4) e così via. Grazie alla ricorsione è possibile realizzare un metodo che calcola il fattoriale applicando esattamente la definizione standard:

```
public int fattoriale(int n) {
  if(n=0)
    return 1;
  else
    return n * fattoriale(n - 1);
}
```

Per eseguire correttamente le procedure ricorsive, l'interprete Java ricorre a una speciale memoria a pila (stack, in inglese), che ha la particolarità di comportarsi come la pila di piatti di un ristorante: il primo piatto che viene preso (tipicamente quello in altro) è anche l'ultimo che vi era stato posto. A causa di questo particolare comportamento, si dice che una pila è una struttura dati di tipo LIFO, acronimo inglese il cui significato è "Last In First Out", ossia "l'ultimo a entrare è il primo a uscire". La ricorsione permette di fornire una soluzione elegante a una serie di problemi, ma proprio a causa del ricorso allo stack, questa eleganza viene pagata con una minore efficienza di esecuzione.

# Costruttore

Il costruttore è uno speciale metodo che ha lo stesso nome della classe e che è privo di valore di ritorno. Il costruttore svolge un compito fondamentale: esso permette infatti di inizializzare i principali attributi di un oggetto durante la fase di creazione, ricorrendo a un'unica operazione:

```
public class Mela {
  private int larghezza;
  private int lunghezza;
  private Color colore;

  public Mela(int lungh , int largh , Color c) {          // Costruttore della classe Mela
    lunghezza = lung;
    larghezza = largh;
    colore = c;
  }
 }
```

Il costruttore svolge un ruolo cruciale nella programmazione a oggetti: grazie a esso, infatti, è possibile mettere in atto le politiche di incapsulamento più rigide. Per esempio, se si crea all'interno di una classe un attributo privato dotato di metodi get ma privo di metodi set, si crea una situazione in cui il costruttore rappresenta l'unica via per inizializzare un simile attributo. Questa tecnica permette di creare attributi immutabili a sola lettura, una scelta che può rivelarsi utile in numerose circostanze.

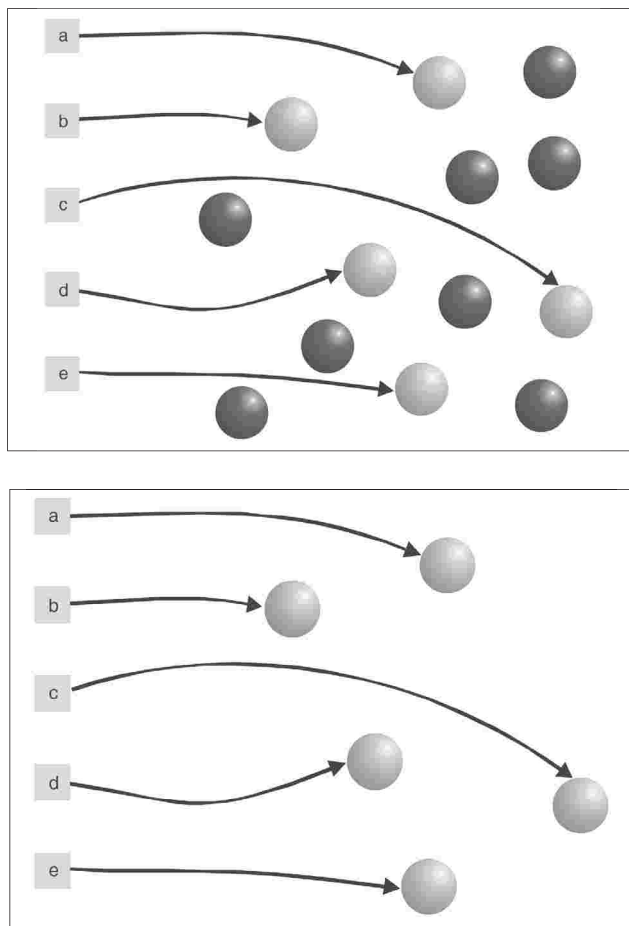
Il costruttore viene invocato in fase di creazione tramite l'operatore `new`; la chiamata deve includere tutti i parametri richiesti:

```
Mela m = new Mela(96,100,new Color("Red"));
```

Ovviamente, non esiste un modo per chiamare il costruttore di una classe in un momento diverso dalla sua creazione. Il costruttore è una componente indispensabile della classe: se il programmatore non ne definisce uno esplicitamente, il compilatore aggiunge automaticamente il costruttore privo di parametri, detto costruttore di default.

## Finalizzatori e garbage collection

Ogni oggetto occupa fisicamente una determinata porzione nella memoria del calcolatore. Dal momento che la memoria è una risorsa finita, è importante capire dove vanno a finire gli oggetti dopo aver svolto il compito per il quale erano stati creati. La gestione della memoria è uno dei punti di forza di Java: la pulizia della memoria dagli oggetti non più utilizzati viene svolta automaticamente durante l'esecuzione da un apposito strumento, detto Garbage Collector (raccoglitore di rifiuti). Non appena la memoria disponibile scende al di sotto di una certa soglia, il Garbage Collector si attiva automaticamente, va in cerca di tutti gli oggetti non più referenziati e li distrugge, liberando la memoria che essi occupavano.

**Figure 6.4 e 6.5 –** *Rappresentazione della memoria prima e dopo l'intervento del Garbage Collector.*

Prima di procedere alla distruzione, il Garbage Collector invoca il metodo finalize() sull'oggetto da rimuovere. Il programmatore può dichiarare tale metodo nelle proprie classi, e inserirvi delle istruzioni da eseguire subito prima della distruzione dell'oggetto:

```
public void finalize() {
  att1 = null;
  att2 = null;
  file.close();
}
```

I finalizzatori sono utili nei contesti in cui la distruzione di una classe comporta operazioni che non vengono compiute automaticamente dal Garbage Collector, come la chiusura di file e di connessioni a database o più in generale il rilascio di risorse di sistema. In tutti gli altri contesti, esso risulta praticamente inutile, e pertanto si sconsiglia di dichiararlo nelle proprie classi.

# Convenzioni di naming

Esistono convenzioni sulle modalità di attribuzione di nomi a variabili, metodi e attributi. Tali convenzioni sono state formalizzate da Sun Microsystems in un documento denominato Code Conventions for the Java Programming Language (http://java.sun.com/docs/codeconv/), che specifica tra l'altro le regole per la disposizione di classi e metodi all'interno di un file, le regole di indentazione e quelle per la stesura dei commenti. Le principali regole di naming sono tre:

- I nomi delle classi devono iniziare con una lettera maiuscola.
- Metodi, variabili e attributi iniziano con una lettera minuscola.
- I nomi composti vengono dichiarati secondo la convenzione CamelCase: le parole vengono riportate per esteso in minuscolo, una di seguito all'altra senza caratteri di separazione, utilizzando un carattere maiuscolo come lettera iniziale di ogni parola.

Queste semplici regole, universalmente utilizzate nel mondo Java, favoriscono una certa uniformità al codice scritto a più mani, e garantiscono un'ottima leggibilità. Ecco di seguito tre identificatori che seguono le convenzioni di naming appena elencate:

```
NomeClasse          nomeMetodo()          nomeVariabile
```

# Ereditarietà

L'ereditarietà è una proprietà fondamentale dei linguaggi orientati agli oggetti. Grazie all'ereditarietà è possibile definire una classe come figlia (o sottoclasse) di una classe già esistente, in modo da estenderne il comportamento. La classe figlia "eredita" tutti i metodi e gli attributi

della superclasse (detta anche classe padre), e in tal modo ne acquisisce il comportamento.

Si provi a immaginare una classe Bicicletta, descritta di seguito in pseudo codice:

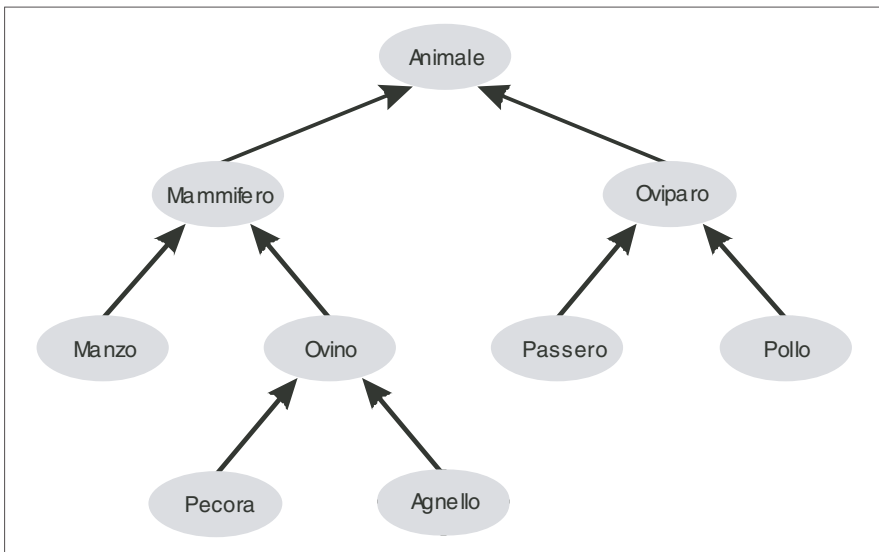
```
public class Bicicletta {

  public Color colore;

  public Color getColore() {
    return colore;
  }
  public void muoviti() {
    sollevaCavalletto();
    muoviPedali();
  }
  public void frena() {
    premiGanasceSuRuota();
  }
  public void curvaDestra() {
    giraManubrioVersoDestra();
  }
  public void curvaSinistra() {
    giraManubrioVersoSinistra();
  }
}
```

Essa è caratterizzata dall'attributo Colore e dai metodi muoviti(), frena(), curvaDestra() e curvaSinistra(). Un motorino è un mezzo simile a una bicicletta (si pensi a un motorino tipo Ciao, dotato di pedali). Esso, tuttavia, è dotato di alcuni attributi aggiuntivi, come la cilindrata e il numero di targa, e di alcuni metodi non presenti nella bicicletta, come per esempio accendiMotore():

```
public class Motorino extends Bicicletta {

  private String targa;
  private int cilindrata;

  public int getCilindrata() {
      return cilindrata;
  }
  public String getTarga() {
    return targa;
  }
  public void accendiMotore() {
    inserisciMiscelaNelCilindro();
    accendiCandela();
  }
}
```

La classe Motorino viene definita, grazie alla direttiva extends, come sottoclasse di Bicicletta. Di conseguenza, la classe Motorino eredita in modo automatico tutti gli attributi e i metodi di Bicicletta, senza la necessità di riscriverli.

L'ereditarietà permette di creare gerarchie di classi di profondità arbitraria, simili ad alberi genealogici, in cui il comportamento della classe in cima all'albero viene gradualmente specializzato dalle sottoclassi. Ogni classe può discendere da un'unica superclasse, mentre non c'è limite al numero di sottoclassi o alla profondità della derivazione.

**Figura 6.6 –** *Un esempio di gerarchia di classi.*

Dal codice di una classe è possibile accedere a metodi e attributi pubblici di qualsiasi superclasse, ma non a quelli privati. Il modificatore protected, che verrà studiato meglio in seguito, permette di definire metodi e attributi accessibili solo dalle sottoclassi.

In Java ogni classe in cui non sia definito esplicitamente un padre, viene automaticamente considerato sottoclasse di Object, che è pertanto il capostipite di tutte le classi Java.

# Overloading

All'interno di una classe è possibile definire più volte un metodo, in modo da adeguarlo a contesti di utilizzo differenti. Due metodi con lo stesso nome possono coesistere in una classe a due condizioni: devono avere lo stesso tipo di ritorno e devono presentare differenze nel numero e nel tipo dei parametri

All'interno di un metodo è sempre possibile invocare un metodo omonimo: spesso, infatti, una famiglia di metodi con lo stesso nome si limita a fornire differenti vie di accesso programmatiche a un'unica logica di base. In un'ipotetica classe Quadrilatero, dotata di un attributo dimensione, è possibile definire tre metodi setter che permettano di impostare l'attributo, specificando un apposito oggetto Dimension, una coppia di interi o nessun parametro per impostare valori di default:

```
Public class Quadrilatero {
  private Dimension dimensione;

  // metodo di base
  public void setSize(Dimension d) {
    dimensione = d;
  }
  // accesso con interi
  public void setSize(int width,int height) {
    SetSize(new Dimension(width,height));
  }
  // impostazione di default
  public void setSize() {
    setSize(new Dimension(100,100));
  }
}
```

In questo esempio si può notare che solamente il primo dei tre metodi setSize() effettua la modifica diretta dell'attributo dimensione; le altre due versioni del metodo si limitano a riformulare la chiamata in modo da renderla adatta al metodo di base.

È possibile effettuare anche l'overloading dei costruttori:

```
public class MyClass {
  private int att1;
  private int att2;

  public MyClass() {
    att1 = 0;
     att2 = 0;
  }
  public MyClass(int a1 , int a2) {
    att1 = a1;
     att2 = a2;
  }
}
```

Per raggiungere il pieno controllo in scenari che prevedano l'overloading di metodi e costruttori, è necessario comprendere l'uso degli identificatori this e super, che verranno illustrati nei prossimi paragrafi.

# Overriding

Grazie all'ereditarietà è possibile estendere il comportamento di una classe sia aggiungendo nuovi metodi sia ridefinendo metodi già dichiarati nella superclasse. Quest'ultima possibilità prende il nome di overriding, ed è un'ulteriore proprietà dei linguaggi a oggetti. Per mettere in atto l'overriding, è sufficiente dichiarare un metodo di cui esiste già un'implementazione in una superclasse: la nuova implementazione acquisirà automaticamente la precedente, sovrascrivendone il comportamento. Nell'esempio del motorino, è naturale pensare alla necessità di fornire una nuova implementazione del metodo muoviti(), in modo tale da adeguarlo allo scenario di un mezzo motorizzato:

```
public class Motorino extends Bicicletta {

  private String targa;
  private int cilindrata;

  public int getCilindrata() {
    return cilindrata;
  }
  public String getTarga() {
   return targa;
  }
  public void accendiMotore() {
   inserisciMiscelaNelCilindro();
   accendiCandela();
  }
  public void muoviti() {
   accendiMotore();
   premiFrizione();
   innestaMarcia(1);
   rilasciaFrizione();
   ....
  }
}
```

La classe Motorino, presente in quest'ultimo esempio, possiede gli stessi metodi e attributi della classe Bicicletta, più alcuni metodi definiti ex novo e una versione nuova del metodo muoviti(), già dichiarato nella superclasse.

# Identificatori speciali this e super

L'identificatore this è un puntatore speciale alla classe che costituisce l'attuale contesto di programmazione. Grazie a this è possibile accedere a qualsiasi metodo o attributo della classe stessa mediante un'espressione del tipo:

```
this.methodo();
```

L'uso di this è indispensabile quando ci si trova a dover distinguere tra un attributo e una variabile con lo stesso nome, come avviene spesso nei metodi setter e nei costruttori:

```
public setAtt1(int att1) {
  this.att1 = att1;  // assegna il valore della variabile locale att1 all'attributo omonimo
}
```

L'identificatore this può essere usato anche per richiamare un costruttore. In questo caso, la parola this deve essere seguita dai parametri richiesti dal costruttore in questione racchiusi tra parentesi, e deve per forza comparire come prima istruzione di un altro costruttore:

```
public class MyClass {
  private int att1;
  private int att2;

  public MyClass() {
    this(0,0);  // chiama il secondo costruttore con i parametri di default
  }
  public MyClass(int a1 , int a2) {
    att1 = a1;
     att2 = a2;
  }
}
```

L'identificatore super ha un uso simile a this ma, a differenza di quest'ultimo, invece di far riferimento alla classe di lavoro si riferisce alla superclasse. Tramite super è possibile invocare la versione originale di un metodo sovrascritto, altrimenti inaccessibile:

```
metodo();                  // chiamata a un metodo della classe
super.metodo();            // chiamata al metodo omonimo presente nella superclasse
```

Spesso i metodi sovrascritti sono estensioni dei metodi equivalenti della superclasse; in questi casi, si può utilizzare super in modo da richiamare la versione precedente del metodo, e quindi aggiungere di seguito le istruzioni nuove:

```
public void metodo1() {
  super.metodo1();  // chiamata a metodo1 della superclasse
  // nuove istruzioni che estendono
  // il comportamento del metodo
  // omonimo della superclasse
}
```

L'uso principale di super è nei costruttori, che devono necessariamente estendere un costruttore della superclasse. Se non viene specificato diversamente, il costruttore di una classe viene considerato estensione del costruttore di default della superclasse (quello privo di parametri). Se si desidera un comportamento differente, o se addirittura la superclasse

non dispone di un costruttore di default, occorre invocare in modo esplicito il supercostruttore desiderato.

```
public class Motorino extends Bicicletta {

  private String targa;
  private int cilindrata;

  public Motorino(Color colore, String targa, int cilindrata) {
                  super(colore);
                  this.targa = targa;
                  this.cilindrata = cilindrata;
  }
  ….
}
```

Anche in quest'ultimo caso, la chiamata al supercostruttore deve precedere qualsiasi altra istruzione.

# Binding dinamico

Ogni classe denota un tipo. Tuttavia, come conseguenza dell'ereditarietà, ogni classe ha come tipo sia il proprio sia quello di tutte le sue superclassi. Grazie a questa proprietà, una classe può essere utilizzata in qualsiasi contesto valido per una qualunque delle sue superclassi. Per esempio, il metodo:

```
public void parcheggia(Bicicletta b) {
  b.muovi();
  b.giraSinistra();
  ….
  b.frena();
}
```

può lavorare sia su oggetti di tipo Bicicletta sia su quelli di tipo Motorino, una cosa abbastanza intuitiva anche nel mondo reale (un parcheggio per automobili può andare bene anche per i taxi, che in fin dei conti sono pur sempre automobili).

Per questo stesso motivo, è possibile formulare una dichiarazione del tipo:

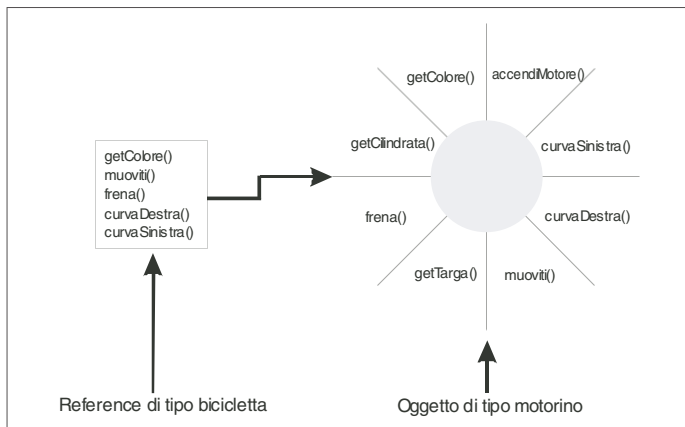
```
Bicicletta b = new Motorino();
```

In questo esempio viene creato un oggetto di tipo Motorino, abbinato a un reference di tipo Bicicletta. Quale effetto produce la chiamata sulla variabile b di un metodo cui sia stato applicato l'overriding? Verrà invocato il metodo definito nella classe padre o quello presente nella sottoclasse?

In Java i metodi sono legati al tipo dell'istanza, non a quello del reference: in altre parole, nonostante il reference sia di tipo Bicicletta, le chiamate avranno sempre l'effetto di invocare il metodo valido per il tipo dell'istanza in questione. Il binding è l'associazione di un metodo alla rispettiva classe: in Java, il binding viene effettuato durante l'esecuzione, in modo tale da garantire che su ogni oggetto venga invocato il metodo corrispondente al tipo. Contrariamente a quanto avviene con linguaggi come il C++, non esiste alcun modo per richiamare su un oggetto un metodo appartenente alla superclasse se questo è stato sovrascritto. Per questa ragione, la chiamata:

```
b.muovi();
```

chiamerà sull'oggetto referenziato dalla variabile b il metodo muovi() di Motorino, dal momento che l'oggetto in questione è di tipo Motorino.

**Figura 6.7 –** *Un reference permette di accedere esclusivamente ai metodi dell'oggetto denotati dal proprio tipo.*

# Upcasting, downcasting e operatore instanceof

Come nei tipi primitivi, l'upcasting, o promozione, è automatico. È sempre possibile referenziare un oggetto con una variabile il cui tipo è quello di una delle sue superclassi:

```
Motorino m = new Motorino();
Bicicletta b = m; // downcasting
```

Quando invece si dispone di un oggetto di un certo tipo referenziato da una variabile di un supertipo, e si desidera passare il reference a un'altra variabile di un tipo più specializzato, è necessario ricorrere all'operatore di casting:

```
 Bicicletta b = new Motorino();
Motorino m = (Motorino)b;
```

Il casting è sempre sconsigliato, dal momento che viola il principio del polimorfismo. Negli scenari in cui si desidera operare un casting, è possibile ricorrere all'operatore instanceof, che permette di verificare il reale tipo di un oggetto:

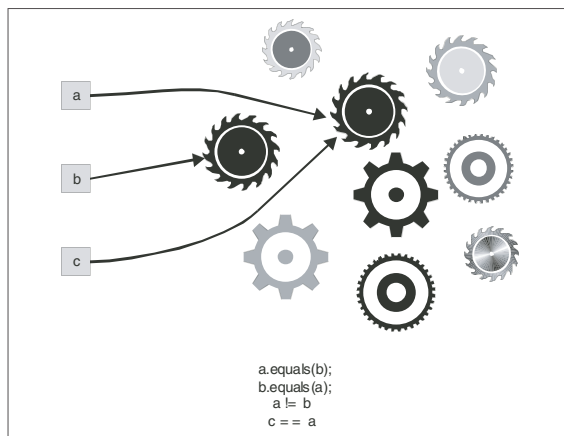
```
If (b instanceof Motorino)
  m  = (Motorino)b;
```

# Equals e operatore ==

Come si è già visto nel caso delle stringhe, quando si lavora con gli oggetti è necessario prestare una grande attenzione a come si usa l'operatore di uguaglianza ==. L'operatore di uguaglianza permette di verificare l'identità tra due reference, che si verifica nel caso in cui essi facciano riferimento allo stesso oggetto in memoria. Qualora si desideri testare la somiglianza tra due oggetti, ossia la circostanza in cui due oggetti distinti presentano lo stesso stato, è necessario ricorrere al metodo equals(). Per esempio, in un caso del tipo:

```
Integer i1 = new Integer(12);
Integer i2 = new Integer(12);
```

Il test i1 == i2 darà esito false, in quanto le due variabili fanno riferimento a due oggetti distinti; al contrario, l'espressione i1.equals(i2) risulterà vera, dal momento che i due oggetti hanno lo stesso identico stato.

**Figura 6.8 –** *Esempi di somiglianza e di identità tra oggetti.*

# Costrutti avanzati

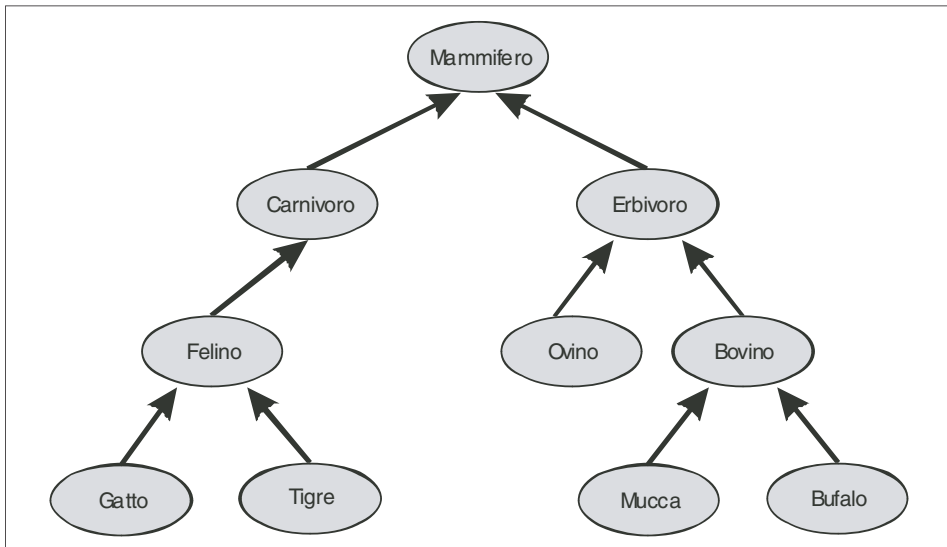
Andrea Gini

## Classi astratte

La classificazione degli oggetti permette di realizzare in modo incrementale entità software dotate di gradi crescenti di specializzazione. Ogni elemento della gerarchia può essere sviluppato e testato in modo indipendente: le migliorie apportate a una classe verranno trasmesse automaticamente a tutte le sottoclassi.

Durante la fase di classificazione capita di creare classi che, pur essendo caratterizzate da attributi e metodi ben precisi, non corrispondono a entità concrete. Nella classificazione degli esseri viventi, per esempio, esiste la categoria dei mammiferi, che racchiude un insieme di attributi comuni a diverse specie viventi (cani, gatti, maiali e così via), ma che di per sé non rappresenta nessun animale. Nella classificazione delle specie animali è possibile trovare ulteriori esempi: in figura 7.1, le categorie Erbivoro, Carnivoro, Felino, Ovino e Bovino non corrispondono ad alcun animale concreto, ma rappresentano comunque passaggi fondamentali nella classificazione.

Nella progettazione del software si presenta spesso un problema simile: le gerarchie molto articolate presentano nodi che corrispondono a categorie astratte di oggetti, indispensabili come passaggi logici di derivazione, ma di fatto non istanziabili. Tali classi vengono dette *astratte* e, a differenza delle classi concrete, possono contenere speciali metodi `abstract`, privi di corpo, che andranno implementati nelle sottoclassi.

**Figura 7.1 –** *Classificazione delle specie viventi.*

Per definire una classe astratta è necessario aggiungere il modificatore abstract, sia nella dichiarazione della classe sia in quella dei metodi privi di implementazione. Nell'esempio seguente viene definita una classe astratta Mammifero, dotata di tre metodi astratti: ingerisci(), digerisci() ed evacua():

```
public abstract class Mammifero {

 public void mangia(Cibo c) {
  ingerisci(Cibo c);
  digerisci();
  evacua();
 }
 public abstract void ingerisci(Cibo c);
 public abstract void digerisci();
 public abstract void evacua();
}
```

Le modalità di ingestione, digestione ed evacuazione sono differenti in ognuna delle sottocategorie: ogni sottoclasse concreta di Mammifero sarà tenuta a fornire un'implementazione di tali metodi, che rifletta la natura particolare dell'entità rappresentata (nei mammiferi carnivori la digestione è sostanzialmente differente rispetto a quella dei mammiferi erbivori ruminanti). Si noti comunque che una classe astratta può contenere metodi concreti, come il metodo mangia nell'esempio, e che tali metodi possono chiamare liberamente i metodi astratti.

# Il contesto statico: variabili e metodi di classe

Gli attributi visti finora sono associati alle istanze di una classe: ogni singola istanza possiede una copia privata di tali attributi, e i metodi della classe lavorano su tale copia privata. Esiste la possibilità di definire attributi statici, ossia variabili legate alla classe, presenti in copia unica e accessibili da tutte le istanze. Allo stesso modo è possibile definire metodi statici, ossia metodi non legati alle singole istanze, che possono operare esclusivamente su variabili statiche. Attributi e metodi statici vengono chiamati anche "attributi e metodi di classe", per distinguerli dagli attributi e dai metodi visti fino a ora, che vengono detti "di istanza".

Metodi e variabili di classe costituiscono il contesto statico di una classe. Per accedere al contesto statico di una classe non è necessario crearne un'istanza; ogni metodo statico è accessibile direttamente mediante la sintassi:

```
NomeClasse.metodoStatico();
```

All'interno di un metodo di classe è possibile accedere solamente a metodi e attributi statici; inoltre, in tale contesto gli identificatori `this` e `super` sono privi di significato.

Gli attributi e i metodi di classe, caratterizzati dal modificatore `static`, sono utili per rappresentare caratteristiche comuni a tutto l'insieme di oggetti appartenenti a una classe. Per esempio, in un'ipotetica classe che denoti le finestre grafiche di un desktop compariranno sia attributi di istanza (come la posizione e la dimensione, che sono caratteristiche di ogni singola finestra) sia attributi di classe (come il colore della barra del titolo, uguale per tutte le finestre presenti nel sistema). Questa situazione può essere rappresentata dal seguente esempio:

```
public class Window {
  private int x;
  private int y;
  private int height ;
  private int width;
  private static Color titleBarColor;

  public void setX(int x) {
     this.x = x;
   }
   ….
   public static void setTitleBarColor(Color c) {
      titleBarColor = c;
   }
}
```

In questo esempio, se si possiede un oggetto di tipo `Window` e si desidera impostarne le dimensioni, uniche per ogni istanza, si provvederà a effettuare una chiamata a metodo come di consueto:

```
w.setWidth(400);
w.setHeight(300);
```

Al contrario, la chiamata:

```
Window.setTitleBarColor(new Color("Red"));
```

va a modificare l'attributo statico titleBarColor, con la conseguenza di modificare tutti gli oggetti di tipo Window presenti in memoria.

---

Le variabili e i metodi statici possono essere richiamati anche tramite l'identificatore di una variabile; pertanto, riferendosi all'esempio precedente, le seguenti istruzioni:

```
Window. setTitleBarColor (new Color("Red"));
w.setTitleBarColor (new Color("Red"));
```

sortiranno il medesimo effetto. Dal momento che metodi e variabili di classe operano esclusivamente sul contesto statico, si preferisce seguire la convenzione di richiamarli unicamente mediante l'identificatore della classe, in modo da sottolineare la differenza.

---

# Interfacce

L'ereditarietà è un mezzo di classificazione straordinario, che tuttavia presenta un grandissimo limite: non esiste un unico criterio di classificazione valido in tutte le circostanze. Nella classificazione degli esseri viventi, tanto per fare un esempio, si incontrano grosse difficoltà a trovare il posto giusto per l'ornitorinco, un animale che abbraccia in modo trasversale i principali criteri di divisione (un mammifero che depone le uova, e possiede caratteristiche morfologiche comuni alla marmotta e alla papera). Nel software queste situazioni di parentela trasversali sono molto comuni, e vengono risolte grazie alle interfacce.

## Interfacce per definire il comportamento

Un'interfaccia è un costrutto che permette di associare un tipo a una collezione di metodi privi di implementazione. A differenza delle classi, le interfacce supportano l'ereditarietà multipla, una caratteristica estremamente importante che distingue le interfacce dalle classi completamente astratte (classi astratte prive di metodi concreti).

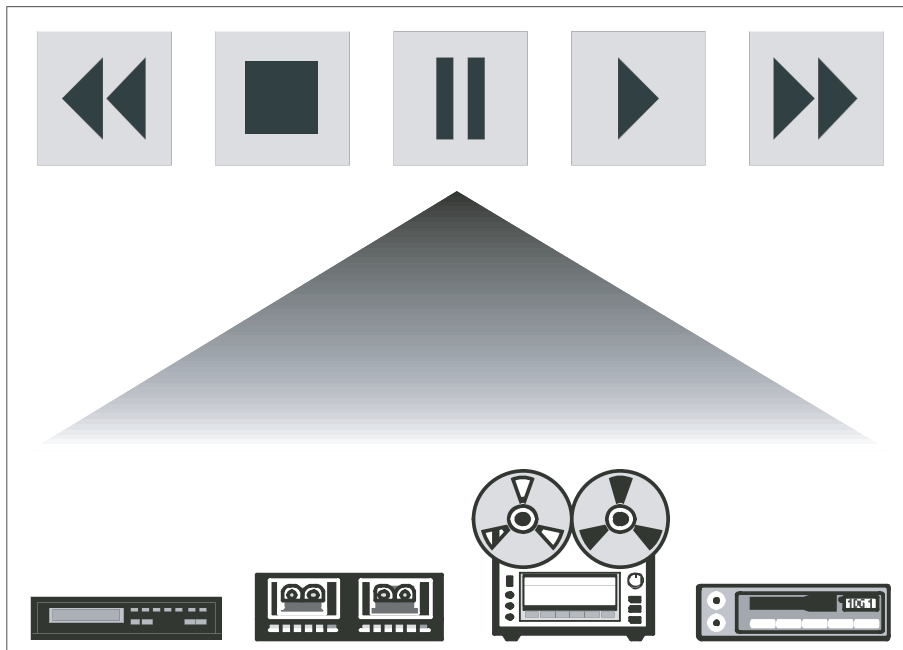
Le interfacce definiscono un protocollo di comportamento che può essere implementato da qualsiasi classe, ovunque si trovi nella gerarchia. I metodi dichiarati all'interno di un'interfaccia non presentano alcun vincolo realizzativo: una classe che implementa un'interfaccia si impegna a fornire un corpo a tutti i metodi definiti dall'interfaccia stessa, al fine di acquisire tale comportamento.

L'uso delle interfacce genera una certa perplessità in chi ha l'abitudine di associare gli oggetti software a una determinata implementazione. Quale può essere l'utilità di un sistema di

classificazione di entità software basato solamente sulle firme dei metodi? La risposta è che l'interfaccia permette di rappresentare una *proprietà* comune a diverse categorie di oggetti, o un *comportamento* presente in oggetti caratterizzati da una differente implementazione.

Un esempio concreto di oggetti diversi dotati di una proprietà in comune può essere trovato in cucina: se durante la preparazione di un dolce la ricetta suggerisce di mischiare gli ingredienti all'interno di un contenitore, si può ricorrere a una marmitta da cucina, realizzata in plastica e con una forma tale da rendere il lavoro di mescolatura particolarmente facile. D'altra parte, in mancanza di una marmitta, è possibile usare qualsiasi altro tipo di contenitore, compresa una pentola per pastasciutta. Nonostante la pentola non sia stata progettata per questo uso, essa ha in comune con la marmitta la proprietà di poter contenere dei fluidi, e per questa ragione potrà essere usata per portare a termine correttamente l'operazione.

Un moderno impianto audio-video presenta invece un esempio lampante di oggetti diversi tra loro dotati della medesima interfaccia utente: CD, DVD, videoregistratore e registratore a cassette sono strumenti completamente diversi dal punto di vista tecnologico. Ciononostante, essi condividono la medesima interfaccia utente (i tasti play, stop, pause, fast forward e rewind). Grazie a questa proprietà, chiunque sappia usare uno qualsiasi di questi strumenti potrà utilizzare senza fatica anche tutti gli altri.

**Figura 7.4 –** *Oggetti molto diversi tra loro possono condividere un comportamento attraverso un'interfaccia*

# Dichiarazione e implementazione di interfacce

Per dichiarare un'interfaccia si ricorre a una sintassi simile a quella usata per le classi:

```
public interface NomeInterfaccia {

  public static final tipo nomeAttributo;

  public  tipo nomeMetodo(tipo par1, tipo par2, …);
}
```

Al posto di class si utilizza la parola chiave interface. Inoltre, i metodi devono per forza essere pubblici e non prevedono un punto e virgola al posto del codice. Nelle interfacce è possibile definire solamente attributi costanti, ossia static e final.

Al pari delle classi, le interfacce possono formare gerarchie, nelle quali è permesso ricorrere all'ereditarietà multipla:

```
public interface MyInterface extends Interface1,Interface2, … {
        ....
}
```

A differenza di quanto avviene per le classi, dove esiste un'unica gerarchia che fa capo a Object, ogni interfaccia può dare vita a una gerarchia di interfacce a sé stante.

Per dichiarare che una classe implementa un'interfaccia, bisogna utilizzare la parola chiave implements nel modo seguente:

```
public class MyClass implements Interface1, Interface2, interface3, ... {
        ....
        corpo della classe
        ....
}
```

dove Interface1, Interface2, Interface3, … sono le interfacce da implementare. È consentita l'implementazione di un numero arbitrario di interfacce.

Se due interfacce contengono metodi con la stessa firma e lo stesso valore di ritorno, la classe concreta dovrà implementare il metodo solo una volta e il compilatore non segnalerà alcun errore. Se i metodi hanno invece lo stesso nome ma firme diverse, la classe concreta dovrà dare un'implementazione per ciascuno dei metodi. Se infine le interfacce dichiarano metodi con lo stesso nome ma con valore di ritorno differente (tipo int getResult() e long getResult()), il compilatore segnalerà un errore, dal momento che il linguaggio Java non permette di dichiarare in una stessa classe metodi la cui firma differisca solo per il tipo del valore di ritorno.

Nel caso degli attributi il problema dei conflitti è molto più semplice: se due interfacce legate

da un qualche grado di parentela dichiarano una costante utilizzando lo stesso nome, sarà sempre possibile accedere all'una o all'altra usando l'identificatore di interfaccia:

```
Interfaccia1.costante;
Interfaccia2.costante;
```

Si noti che in questo caso le costanti omonime possono anche essere di tipo diverso.

## Un esempio concreto

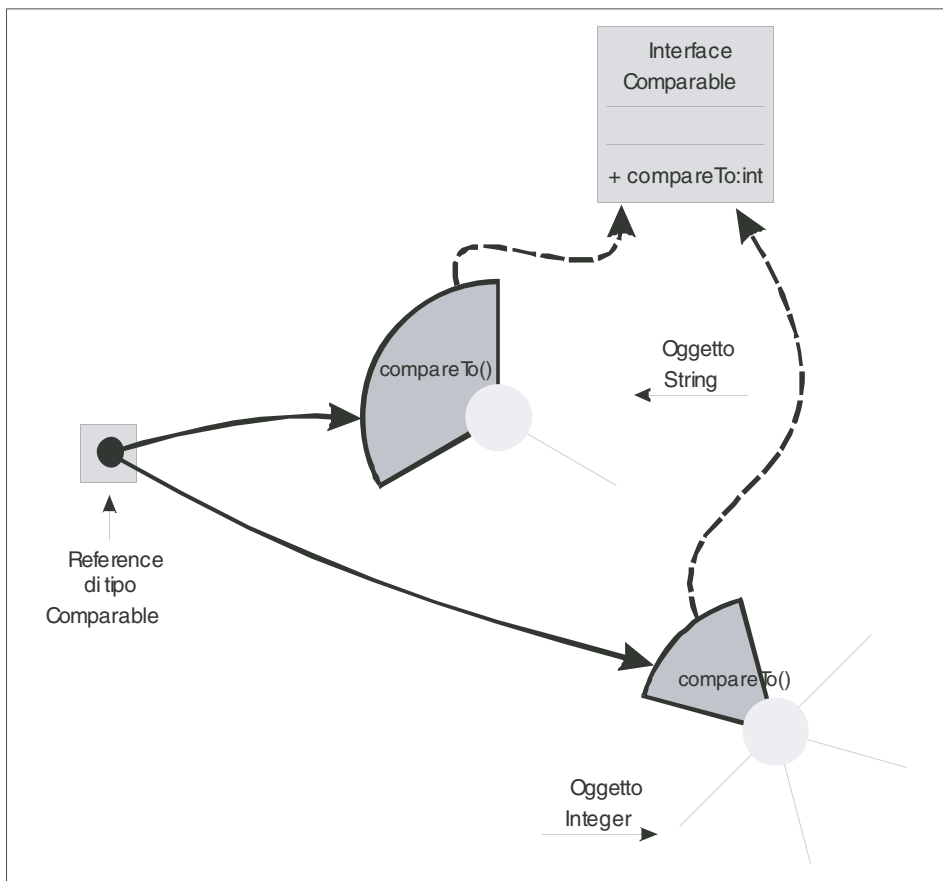
Per non restare troppo nell'astratto, ecco un esempio di reale utilità. Le API Java definiscono l'interfaccia Comparable:

```
public interface Comparable {
  public int compareTo(Object o);
}
```

Tale interfaccia viene implementata da un gran numero di classi molto diverse tra loro: BigDecimal, BigInteger, Byte, ByteBuffer, Character, CharBuffer, Charset, CollationKey, Date, Double, DoubleBuffer, File, Float, FloatBuffer, IntBuffer, Integer, Long, LongBuffer, ObjectStreamField, Short, ShortBuffer, String e URI.

Le classi appena elencate hanno in comune tra di loro soltanto il fatto di essere ordinabili. Dal momento che implementano tutte l'interfaccia Comparable, è possibile scrivere un metodo che permetta di ordinare un array di oggetti di qualsiasi tipo tra quelli elencati:

```
public class ComparableSorter {

  public static Comparable[] sort(Comparable[] list) {
    for(int i = 0 ; i < list.length ; i++ ) {
      int minIndex = i;
      for(int j = i ; j < list.length ; j++) {
        if ( list[j].compareTo(list[minIndex]) < 0  )
          minIndex = j;
      }
      Comparable tmp = list[i];
      list[i] = list[minIndex];
      list[minIndex] = tmp;
    }
    return list;
  }
}
```

**Figura 7.2 –** *Le interfacce permettono di operare in modo uniforme su oggetti diversi tra loro.*

Il metodo ordina() non è interessato a quale sia il tipo concreto degli oggetti che gli vengono passati: l'unico requisito che gli interessa è che essi implementino l'interfaccia Comparable, in modo da permettere l'esecuzione dell'algoritmo di ordinamento. Dal punto di vista del metodo ordina(), un vettore di Integer è uguale a un vettore di String: il suo comportamento non è influenzato da questa differenza.

Questo metodo funziona su tutti gli oggetti che implementano l'interfaccia Comparable, persino su oggetti che al momento non esistono. Chi realizza le classi concrete ha la responsabilità di stabilire un criterio di confronto e di incorporarlo nel metodo compareTo(): la logica di ordinamento presente nel metodo ordina() trascende il particolare criterio adottato per l'oggetto concreto.

# Tipi e polimorfismo

Il polimorfismo è un'importante proprietà dei linguaggi a oggetti: attesta la possibilità di utilizzare un oggetto al posto di un altro, laddove esista una parentela tra i due. Grazie alle interfacce è possibile esprimere a un livello di dettaglio molto profondo l'appartenenza a determinate categorie, e creare procedure in grado di operare in modo trasversale su un gran numero di oggetti accomunati solo da una proprietà o da un comportamento.

Come si è già constatato in precedenza, una classe ha come tipo quello della propria classe e di tutte le sue superclassi. L'interfaccia denota a sua volta un tipo: pertanto, una classe ha tanti tipi quante sono le interfacce implementate (comprese le eventuali super interfacce). Java è un linguaggio *strong typed*: il legame tra un oggetto e i suoi tipi è un aspetto fondamentale e inderogabile, al contrario di linguaggi come C o C++ dove il legame tra tipo e oggetto è lasco, e sono consentite anche operazioni di casting prive di senso. In Java è obbligatorio definire esplicitamente il tipo di una variabile; inoltre, il casting tra oggetti funziona solamente se il tipo dell'oggetto coincide con ciò che viene richiesto dall'operatore di casting. Una buona norma di programmazione è quella di manipolare gli oggetti utilizzando una variabile del tipo dotato dei requisiti più stringenti in relazione al contesto. In questo modo, si garantisce il massimo grado di riutilizzo a ogni singolo elemento del sistema.

# Classi e interfacce interne

Java permette di definire classi e interfacce all'interno di altre classi, con un livello di nidificazione arbitrario:

```
public class MyClass {

  public void metodoOmonimo() {
    ….
  }
public interface MyInnerInterface {
  public void m1();
  public void m2();
}
public class MyInnerClass {
  public void metodoOmonimo() {
    ….
  }
}
}
```

Classi e interfacce interne possono essere richiamate nella classe di livello superiore utilizzando il loro nome, senza particolari differenze rispetto al modo in cui si utilizza una qualsiasi altra classe presente nel name space del sorgente. Di contro, se si desidera accedere a esse dal-

l'esterno del sorgente in cui sono state dichiarate, è necessario utilizzare il percorso completo ricorrendo alla dot notation:

```
MyClass.MyInnerClass m = new MyClass.MyInnerClass();
```

Nel codice di esempio, si può notare che la classe interna definisce un metodo utilizzando lo stesso nome di un metodo della classe di livello superiore. In casi come questo, qualora il programmatore desideri chiamare il metodo omonimo della classe di livello superiore, dovrà accedere all'istanza della classe contenitore mediante un uso particolare di this, che viene richiamato come se fosse un attributo accessibile attraverso l'identificatore corrispondente al nome della classe esterna:

```
MyClass.this. metodoOmonimo();
```

Le classi interne sono state introdotte a partire dal JDK 1.1, in particolare per supportare il modello di eventi tipico delle interfacce grafiche. Pertanto esse verranno trattate in modo approfondito solamente in tale contesto.

# I package

Il linguaggio Java offre la possibilità di organizzare le classi in package, uno strumento di classificazione gerarchico simile alle directory di un file system. Un package è un contenitore che può raccogliere al suo interno sia classi sia altri package, secondo una struttura gerarchica. Grazie ai package è possibile suddividere un software in moduli, raggruppando insiemi di classi che svolgono un determinato compito. Una simile forma di organizzazione diventa indispensabile nei moderni sistemi software, composti di solito da centinaia o migliaia di classi.

L'organizzazione delle classi in package permette inoltre di risolvere il problema del conflitto di nomi (in inglese name clash). Il conflitto di nomi è un problema molto sentito nella comunità dei programmatori a oggetti: all'interno di progetti di grandi dimensioni, la necessità di dare un nome a ogni cosa conduce a dover riutilizzare più volte nomi comuni tipo "Persona", "Cliente" o "Studente". La divisione in package risolve il problema del conflitto di nomi, dal momento che in questo caso è sufficiente che i nomi siano unici all'interno di un package.

## Dichiarazione di package

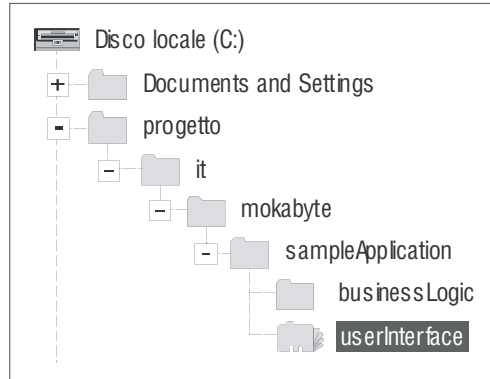
Per includere una classe in un package è necessario inserire in testa al sorgente la dichiarazione:

```
package nomepackage;
```

Se si desidera inserire la classe in un sottopackage, bisogna utilizzare la dot notation:

```
package pakage1.package2.myPackage;
```

Oltre alla dichiarazione di appartenenza, è necessario che il file sorgente venga posto fisicamente all'interno di una struttura di directory analoga a quella dei package sottostanti, a partire da una locazione che prenderà il nome di radice. Si osservi la figura 7.3:

**Figura 7.3 –** *Esempio di organizzazione in package all'interno di un file system.*

Se si volesse creare una classe MainFrame all'interno del package it.mokabyte.sampleApplication .userInterface, sarebbe necessario creare nella directory C:/progetto/it/mokabyte/sampleApplication/ userInterface un sorgente "MainFrame.java" dotato di un'intestazione di questo tipo:

```
package it.mokabyte.sampleProject.userInterface;

public class MainFrame {
....
}
```

In uno scenario come questo, la directory "progetto" svolge il ruolo di radice della gerarchia: la posizione dei package all'interno dell'albero viene considerata in relazione a quest'ultima.

## Compilazione ed esecuzione

L'organizzazione di un progetto in package richiede una certa attenzione in fase di compilazione ed esecuzione, al fine di evitare errori comuni che generano una grossa frustrazione nei principianti.

In primo luogo, per identificare univocamente una classe è necessario specificarne per esteso il nome assoluto, comprensivo di identificatori di package:

```
it.mokabyte.sampleApplication.userInterface.MainFrame
```

In secondo luogo, prima di invocare i comandi bisogna posizionarsi sulla radice dell'albero di package; in caso contrario, il compilatore e la JVM non riusciranno a trovare i file. Riprendendo l'esempio precedente, prima di compilare o eseguire la classe MainFrame sarà necessario effettuare una chiamata di questo tipo:

```
cd c:/progetto
```

Per compilare la classe MainFrame.java bisogna digitare:

```
javac it\mokabyte\sampleApplication\userInterface\MainFrame.java
```

su piattaforma Windows e:

```
javac it/mokabyte/sampleApplication/userInterface/MainFrame.java
```

in Unix-Linux.

Per eseguire il main della classe, invece, bisogna invocare il comando Java specificando il nome assoluto:

```
java it.mokabyte.sampleApplication.userInterface.MainFrame
```

Quando si comincia a lavorare su progetti di grandi dimensioni organizzati in package, è bene prendere l'abitudine di separare i file sorgenti dalle classi in uscita dal compilatore. Il flag –d del compilatore javac permette di specificare la directory di destinazione del compilatore; pertanto, se si desidera compilare la classe MainFrame in modo tale da porre i file .class nella directory C:\classes, si dovrai invocare il comando:

```
javac -d C:¥classes it\mokabyte\sampleApplication\userInterface\MainFrame.java
```

All'interno della directory C:\classes verranno automaticamente generate le cartelle corrispondenti ai package, con i file .class nelle posizioni corrette.

## Dichiarazione import

Per accedere alle classi contenute in un package è necessario specificare il nome della classe per esteso, ricorrendo alla dot notation sia in fase di dichiarazione sia di assegnamento:

```
it.mokabyte.sampleApplication.userInterface.MainFrame m =
                              new it.mokabyte.sampleApplication.userInterface.MainFrame();
```

La scomodità di tale approccio è evidente: l'uso di nomi composti così lunghi può generare altrettanti problemi quanto il conflitto di nomi. Per non essere costretti a specificare ogni volta tutto il percorso verso la classe MainFrame, è possibile importare la classe all'interno del name space del sorgente su cui si sta lavorando, aggiungendo un'apposita clausola import all'intestazione del file:

```
import it.mokabyte.sampleApplication.userInterface.MainFrame;

public class MyClass {
  public static void main(String argv[]) {
      MainFrame f = new MainFrame();
  }
}
```

Grazie alla import, l'identificatore MainFrame entra a far parte dello spazio dei nomi (name space) del sorgente; sarà compito del compilatore associare il nome:

```
MainFrame
```

con la classe:

```
it.mokabyte.sampleApplication.userInterface.MainFrame
```

La import viene spesso utilizzata per importare interi package, utilizzando il carattere * al posto del nome della classe:

```
import it.mokabyte.sampleApplication.userInterface.*;
```

In questo caso, il name space del sorgente comprenderà i nomi di tutte le classi contenute nel package importato. Naturalmente, l'import di interi package può a sua volta generare conflitti sui nomi, dal momento che package differenti possono contenere nomi uguali. Queste eventualità verranno in ogni caso segnalate in fase di compilazione, e potranno essere risolte ricorrendo, solo nei casi di conflitto, all'uso di nomi completi.

## Convenzioni di naming dei package

I nomi dei package seguono la consueta convenzione CamelCase, con iniziale minuscola. Per l'organizzazione in sottopackage, le specifiche Sun consigliano di seguire la regola del nome di dominio invertito: per esempio, IBM (dominio ibm.com) inserisce le proprie classi in package che hanno come prefisso com.ibm; allo stesso modo, Sun Microsystems (sun.com) utilizza il prefisso com.sun. Grazie a questa convenzione, è possibile garantire l'unicità dei nomi di classe senza il bisogno di ricorrere a una qualche autorità centralizzata.

## Principali package del JDK

A differenza di altri linguaggi di programmazione, le classi di sistema di Java non sono riunite in librerie, ma in package. Ogni package contiene classi che permettono di svolgere un determinato compito: grafica, comunicazione in rete, gestione del file system e così via. Di seguito, vengono presentati i package più importanti del linguaggio:

- java.lang: classi base del linguaggio, tra le quali si trovano Object, String, le wrapper class (Integer, Boolean ecc.) e la classe Math (che contiene le più importanti funzioni matematiche). A differenza degli altri package, java.lang non deve essere importato, dato che fa già parte della dotazione standard del linguaggio.
- java.io: contiene tutte le classi necessarie a gestire il file system e l'input output.
- java.util: classi di utilità, come Vector, Hashtable o Date.
- java.net: qui si trovano tutte le classi di supporto alla comunicazione via rete.
- java.awt: toolkit grafico.
- javax.swing: classi per la gestione della grafica a finestre.

L'uso delle classi presenti in questi e altri package verrà chiarito nei prossimi capitoli.

# Modificatori

I modificatori sono parole riservate del linguaggio che permettono di impostare determinate caratteristiche di classi, metodi e attributi. Alcuni di questi modificatori, come abstract e static, sono già stati trattati ampiamente nei paragrafi precedenti; altri, come public, private e protected, verranno ora illustrati in modo completo e formale.

## Moficatori di accesso

I modificatori di accesso permettono di impostare il livello di visibilità dei principali elementi di un sorgente Java, ossia classi, metodi e attributi. A ognuno di questi elementi è possibile assegnare uno dei seguenti livelli di protezione:

- private: l'elemento è visibile solo all'interno di una classe.
- nessun modificatore (package protection): l'elemento è visibile a tutte le classi che fanno parte dello stesso package
- protected: l'elemento è accessibile a tutte le classi del package e a tutte le sottoclassi, anche se dichiarate in package differenti.
- public: l'elemento è visibile ovunque.

Il modificatore private è caratteristico degli attributi: il principio dell'incapsulamento raccomanda di dichiarare private tutti gli attributi, e di consentirne l'accesso in modo programmatico

mediante metodi getter. Ci sono comunque diverse occasioni in cui conviene dichiarare private anche un metodo, qualora esso venga chiamato solamente all'interno della classe.

La package protection viene utilizzata soprattutto per classi che hanno una loro utilità all'interno di un package, ma che si desidera restino nascoste al resto del sistema.

Si ricorre a protected in tutte le occasioni in cui si desidera rendere un metodo visibile solo alle sottoclassi: tipicamente, vengono dichiarati protected i metodi getter relativi ad attributi che si desidera restino inaccessibili al di fuori del dominio della classe e delle sue sottoclassi.

Infine, il modificatore public rende un elemento visibile ovunque: si raccomanda di utilizzare tale modificatore con parsimonia e di limitarne l'uso a classi e metodi. Si ricordi che l'insieme dei metodi pubblici di una classe è l'interfaccia attraverso la quale una classe comunica con il mondo esterno. Pertanto, bisogna scegliere con cura quali metodi rendere pubblici e quali no, avendo la stessa accortezza del progettista di elettrodomestici nel non lasciare fili o ingranaggi scoperti.

## Final

Il modificatore final assume un significato diverso a seconda del contesto di utilizzo. Se è abbinato a un attributo, esso lo rende immutabile. Tale circostanza impone che l'assegnamento di una variabile final venga effettuato nello stesso momento della dichiarazione. Spesso final viene utilizzato in abbinamento a static per definire delle costanti:

```
public static final float pigreco = 3.14;
```

Bisogna fare attenzione al fatto che, quando si dichiara final un reference a un oggetto, a risultare immutabile è il reference e non l'oggetto in sé. Se si definisce un attributo del tipo:

```
public static final Vector list = new Vector();
```

L'uso di final vieta l'operazione di assegnamento sulla variabile:

```
list = new Vector() // errore: la variabile list è final
```

Al contrario, è consentito chiamare i metodi che modificano lo stato dell'oggetto:

```
list.add("Nuovo Elemento");
```

L'uso di final in abbinamento a un metodo ha invece l'effetto di vietarne l'overriding nelle sottoclassi; se associato a una classe ha l'effetto di proibire del tutto la creazione di sottoclassi (in una classe final, tutti i metodi sono final a loro volta). L'uso di final su metodi e classi, oltre ad avere dei ben precisi contesti di utilizzo, presenta l'ulteriore vantaggio di permettere al compilatore di effettuare ottimizzazioni, rendendo l'esecuzione più efficiente. Per questa ragione alcune classi di sistema, come String e StringBuffer, sono state definite final.

## Native

Il modificatore native serve a dichiarare metodi privi di implementazione. A differenza dei metodi abstract, i metodi native vanno implementati in un qualche linguaggio nativo (tipicamente in C o C++). La filosofia di Java non incoraggia la dichiarazione di metodi nativi, ossia dipendenti dalla macchina. Per questa ragione l'effettiva creazione di metodi nativi risulta piuttosto macchinosa.

## Strictfp

Il modificatore strictfp, utilizzabile sia nei metodi sia nelle classi, ha lo scopo di forzare la JVM ad attenersi strettamente allo standard IEEE 754 nel calcolo di espressioni in virgola mobile. Java, infatti, utilizza di norma una variante di tale standard che offre maggiore precisione. Vi sono tuttavia situazioni in cui è necessario fare in modo che i risultati delle operazioni siano assolutamente conformi allo standard industriale, anche se questo implica una minore precisione.

## Transient, volatile e synchronized

Il modificatore transient permette di specificare che un determinato attributo non concorre a definire lo stato di un oggetto. Esso verrà studiato in profondità nei capitoli relativi alla serializzazione e a JavaBeans. L'uso di volatile e synchronized, due modificatori dotati di una semantica piuttosto complessa, verrà invece illustrato nel capitolo sui thread e sulla concorrenza.

Capitolo 8

# Eccezioni

ANDREA GINI

Durante la normale esecuzione, un programma può andare incontro a vari problemi di funzionamento. Tali problemi, a volte, non dipendono dal codice, ma da eventi del mondo reale che non sono sotto il controllo del programmatore. Si pensi a un programma che legge un file dal disco: se durante l'esecuzione il disco si rompe, il programma andrà incontro a un fallimento. Tale fallimento non è dovuto a un errore di programmazione: si tratta semplicemente di uno di quegli incidenti che nel mondo reale possono capitare quando meno ce lo si aspetta.

Java definisce in modo rigoroso il concetto di eccezione, e prevede un apposito costrutto per favorirne la gestione. A differenza di C++, la gestione delle eccezioni in Java non è derogabile: tutte le situazioni in cui può avvenire un'eccezione devono essere gestite in modo esplicito dal programmatore.

Questo approccio rende i programmi di gran lunga più robusti, e riduce notevolmente i problemi di affidabilità e di portabilità del codice.

## Errori ed eccezioni

I problemi che si possono presentare in fase di esecuzione appartengono a due categorie: errori di runtime ed eccezioni.

Gli errori di runtime si verificano quando un frammento di codice scritto correttamente si trova a dover gestire una situazione anomala, che impedisce di proseguire l'esecuzione. Si osservi il seguente assegnamento:

```
a = b / c;
```

L'istruzione è formulata correttamente e funzionerà senza problemi in quasi tutti i casi; tuttavia, se per qualche ragione la variabile c dovesse assumere il valore 0, l'assegnamento non potrebbe avere luogo, dal momento che la divisione per 0 non è definita. Si noti che le circostanze per cui la variabile c potrebbe assumere il valore 0 non dipendono necessariamente dallo sviluppatore: se il programma legge i dati da un file, per esempio, potrebbe capitare che quest'ultimo sia stato formulato in modo sbagliato dall'operatore incaricato di inserire i dati. La caratteristica principale degli errori di runtime è che essi provocano quasi sempre la chiusura irrevocabile del programma.

Le eccezioni sono situazioni anomale che interrompono un'operazione durante il suo normale svolgimento. Se un computer sta dialogando con un server tramite la rete, e sul più bello un fulmine incenerisce la linea di collegamento, il programma client dovrà affrontare una circostanza accidentale e imprevista, che in molti casi potrà essere gestita senza provocare la chiusura del programma. Chiunque navighi in Internet si sarà trovato almeno una volta nell'impossibilità di collegarsi a un sito: in casi come questo, il browser si limita a presentare un messaggio di errore a schermo, quindi si predispone a ricevere nuove direttive dall'utente.

# Gestione delle eccezioni

Molti oggetti Java possono generare eccezioni durante la chiamata di un loro metodo o addirittura durante la creazione: questa esigenza è specificata dalla documentazione della classe. Il tentativo di aprire un file in lettura, per esempio, può causare una FileNotFoundException se file che si desidera aprire non esiste. Se si prova a compilare un programma che contiene un'istruzione come la seguente:

```
Reader i = new FileReader("file.txt");
```

Il compilatore segnalerà la necessità di "catturare" (catch) in modo esplicito l'eccezione FileNotFoundException:

```
C:\program.java:6: unreported exception java.io.FileNotFoundException;
 must be caught or declared to be thrown Reader i = new FileReader("file.txt");
^
1 error
```

L'esempio seguente mostra la sintassi da utilizzare nella situazione appena descritta:

```
try {
  BufferedReader i = new BufferedReader(new FileReader("text.txt"));
}
catch(FileNotFoundException fnfe) {
  System.out.println("Il file indicato non è stato trovato");
  fnfe.printStackTrace();
}
```

In questo caso, il computer *prova* (`try`) a eseguire l'istruzione contenuta nel primo blocco. Se durante il tentativo si verifica un'eccezione, quest'ultima viene *catturata* (`catch`) dal secondo blocco, che stampa a schermo un messaggio di errore e mostra lo stack di esecuzione del programma:

```
Il file indicato non è stato trovato
java.io.FileNotFoundException: file.txt (Impossibile trovare il file specificato)
        at java.io.FileInputStream.open(Native Method)
        at java.io.FileInputStream.<init>(FileInputStream.java:103)
        at java.io.FileInputStream.<init>(FileInputStream.java:66)
        at java.io.FileReader.<init>(FileReader.java:41)
        at Untitled.main(Untitled.java:7)
```

Se invece non sorgono problemi, il runtime Java ignora il contenuto nel blocco `catch` e prosegue nella normale esecuzione.

## Costrutto try – catch – finally

Il costrutto generale per la gestione delle eccezioni ha la seguente forma:

```
try {
        istruzione1();
   istruzione2();
   ....
}
catch(Exception1 e1) {
        // gestione dell'eventuale problema nato nel blocco try
}
catch(Exception2 e2) {
        // gestione dell'eventuale problema nato nel blocco try
}
finally {
        // codice da eseguire comunque al termine del blocco try
}
```

Il blocco `try` contiene un insieme di istruzioni che potrebbero generare eccezioni. Generalmente, si racchiude all'interno di tale blocco un'intera procedura: se durante l'esecuzione della stessa una qualsiasi delle istruzioni genera un'eccezione, il flusso di esecuzione si interrompe e la gestione passa al blocco `catch` incaricato di gestire l'eccezione appena sollevata.

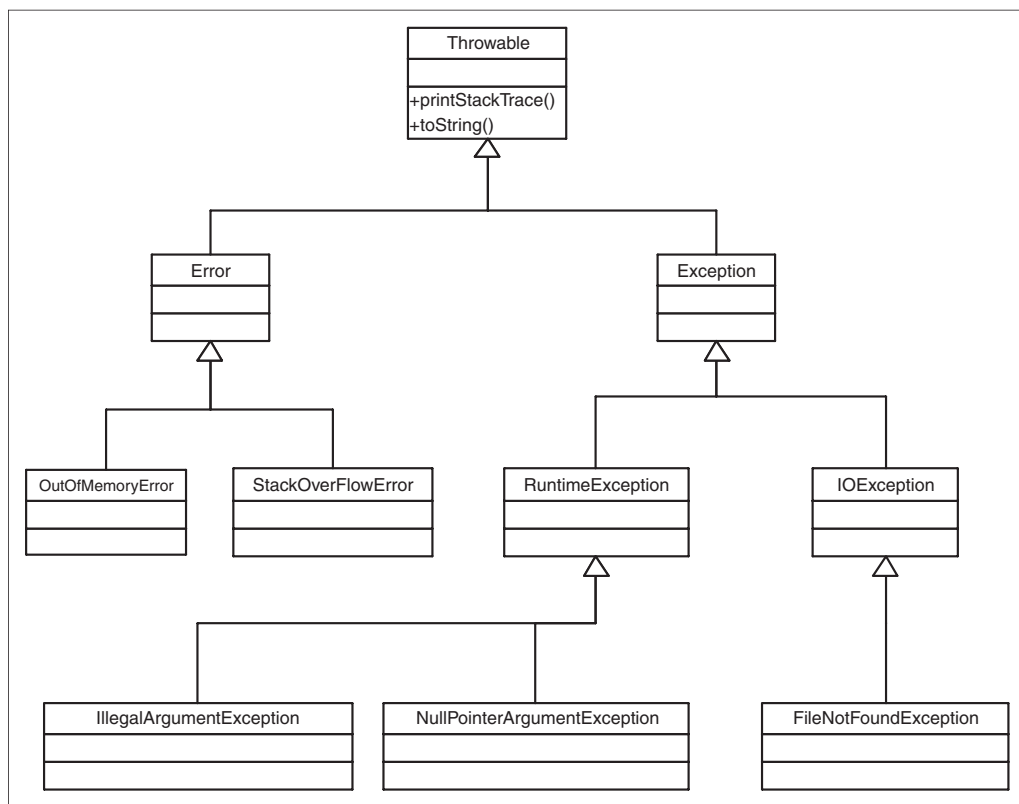
Il blocco `catch` specifica quale eccezione si desidera gestire e quali istruzioni eseguire in quella circostanza. È possibile ripetere più volte il blocco `catch`, in modo da permettere una gestione differenziata delle eccezioni generate dal blocco `try`. In Java è obbligatorio inserire un `catch` per ogni possibile eccezione, anche se naturalmente si può lasciare in bianco il blocco corrispondente.

La clausola finally contiene un blocco di istruzioni da eseguire comunque dopo il blocco try, sia se esso è terminato senza problemi sia nel caso abbia sollevato una qualche eccezione.

## Gerarchia delle eccezioni

Le eccezioni sono oggetti e hanno una loro gerarchia. In cima a essa si trova l'oggetto Throwable, che definisce costruttori e metodi comuni a tutte le sottoclassi. Tra questi metodi, vale la pena di segnalare i seguenti, che permettono di stampare a schermo o su file informazioni diagnostiche:

- printStackTrace(): stampa su schermo lo stack di sistema, segnalando a quale punto del flusso di esecuzione l'eccezione è stata generata e come si è propagata.
- toString(): produce una stringa con le informazioni che caratterizzano l'eccezione.

**Figura 9.1 –** *Gerarchia delle eccezioni Java.*

La classe Throwable dà origine a una gerarchia enorme (il JDK 1.4 contiene 330 tra eccezioni ed errori). Le eccezioni possono essere suddivise in due categorie: checked e unchecked. Le prime devono obbligatoriamente essere gestite all'interno di un blocco try - catch: oltre alla generica Exception, esistono eccezioni che segnalano malfunzionamenti di input output, problemi di rete, errori nella formattazione dei dati e così via.

La seconda famiglia, che comprende gli Error, le RuntimeException e le relative sottoclassi, non obbliga il programmatore a ricorrere a una try - catch, dal momento che queste eccezioni segnalano i malfunzionamenti per i quali non è previsto recupero. Si notino OutOfMemoryError e StackOverflowError, due condizioni che tipicamente segnalano l'esaurimento delle risorse macchina e la necessità di terminare il programma.

L'organizzazione gerarchica delle eccezioni consente una gestione per categorie, secondo il principio della genericità. Per esempio, se viene generata IOException, può essere gestita sia con un'istruzione del tipo:

```
catch(IOException ioe)
```

sia con una più generica:

```
catch(Exception e)
```

È anche possibile predisporre una gestione delle eccezioni per ordine crescente di genericità. Nell'esempio seguente, i primi tre catch gestiscono eccezioni ben precise, mentre l'ultimo gestisce tutte le eccezioni che non sono state gestite dai blocchi precedenti:

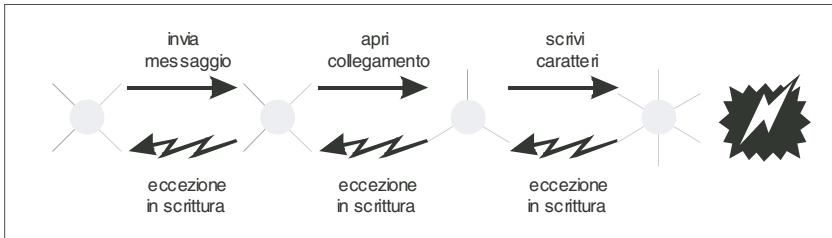
```
try {
  ....
}
catch(NullPointerException npe) {}
catch(FileNotFoundException fnfe) {}
catch(IOExcetpion ioe) {}
catch(Exception e) {}
```

Si noti che la gestione di FileNotFoundException deve precedere quella di IOException, dato che la prima è una sottoclasse della seconda. Per la stessa ragione, la gestione di Exception deve per forza comparire in fondo all'elenco.

## Propagazione: l'istruzione throws

Il costrutto try – catch permette di gestire i problemi localmente, nel punto preciso in cui sono stati generati.

Tuttavia, in un'applicazione adeguatamente stratificata, può essere preferibile fare in modo che le classi periferiche lascino rimbalzare l'eccezione verso gli oggetti chiamanti, in modo da delegare la gestione dell'eccezione alla classe che possiede la conoscenza più dettagliata del sistema.

**Figura 13.2 –** *Propagazione di un'eccezione in un sistema stratificato.*

L'istruzione throws, se è presente nella firma di un metodo, consente di propagare l'eccezione al metodo chiamante, in modo da delegarne a esso la gestione:

```
public void sendMessage(String message) throws IOException {
  // istruzioni che possono generare una IOException
}
```

Naturalmente, la chiamata al metodo sendMessage, definito con clausola throws, dovrà essere posta all'interno di un blocco try - catch. In alternativa, il metodo chiamante potrà a sua volta delegare la gestione dell'eccezione mediante una throws.

## Lancio di eccezioni: il costrutto throw

Finora si è visto come gestire metodi che possono generare eccezioni, o in alternativa come delegare la gestione delle stesse a un metodo chiamante. Ma cosa si deve fare se si desidera *generare* in prima istanza un'eccezione? Si provi a definire un metodo per il calcolo del fattoriale: per creare una procedura robusta, è necessario segnalare un errore nel caso si provi a effettuare una chiamata con un parametro negativo, dal momento che la funzione fattoriale non è definita in questo caso. Il sistema ideale per segnalare l'irregolarità di una simile circostanza è la generazione di una IllegalArgumentException, come nell'esempio seguente:

```
public class LibreriaMatematica  {
static int fattoriale(int n) throws IllegalArgumentException {
 if(n<0)
   throw new IllegalArgumentException("Il parametro deve essere positivo");

   long f = 1;
   while ( n > 0 ) {
     f = f * n;
     n = n - 1;
   }
   return f;
 }
}
```

A questo punto, le chiamate al metodo fattoriale(int n) devono necessariamente comparire all'interno di un blocco try – catch:

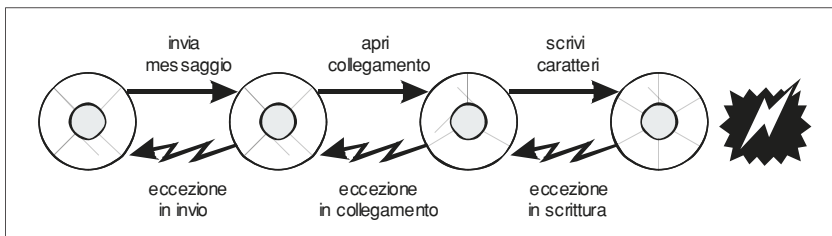
```
try {
  String stringNum = JOptionPane.showInputDialog(null, "Inserisci un numero");
  int num = Integer.parseInt(stringNum);
  long res = LibreriaMatematica.fattoriale(num);
  System.out.println("Il fattoriale di " + num + " è " + res);
}
catch (IllegalArgumentException e) {
  System.out.println(e.getMessage());
}
```

L'istruzione throw richiede come argomento un oggetto Throwable o una sua sottoclasse. È possibile utilizzare throw all'interno di un blocco catch, qualora si desideri ottenere sia la gestione locale di un'eccezione sia il suo inoltro all'oggetto chiamante:

```
public void sendMessage(String message) throws IOException {
  try {
    // istruzioni che possono generare una IOException
  }
  catch(IOException ioe) {
    System.out.println("IOException nel metodo sendMessage");      // gestione locale
    throw ioe;                                                      // inoltra l'eccezione al chiamante
  }
}
```

## Catene di eccezioni

Il meccanismo di inoltro delle eccezioni descritto nel paragrafo precedente può essere ulteriormente raffinato grazie a una funzionalità introdotta a partire da Java 1.4: le eccezioni concatenate. In un sistema adeguatamente stratificato, capita che un'eccezione catturata a un certo livello sia stata generata a partire da un'altra eccezione, nata da un livello più profondo. L'uso di differenti livelli di astrazione rende spesso incomprensibile un'eccezione di livello più basso:

**Figura 13.3 –** *Uno scenario di generazione di eccezioni a catena.*

Per questa ragione, a partire da Java 1.4 le eccezioni permettono, in fase di creazione, di specificare una *causa*, ossia l'oggetto Throwable che ha provocato l'eccezione al livello più alto. Questo meccanismo può dar vita a vere e proprie catene di eccezioni, che forniscono una diagnostica molto dettagliata, utile a comprendere la vera natura di un problema.

## Eccezioni definite dall'utente

Nonostante l'enorme varietà di eccezioni già presenti in Java, il programmatore può facilmente crearne di proprie, qualora desideri segnalare condizioni di eccezione tipiche di un proprio programma. Per creare una nuova eccezione è sufficiente dichiarare una sottoclasse di Exception (o di una qualsiasi altra eccezione esistente) e ridefinire uno o più dei seguenti costruttori:

- Exception(): crea un'eccezione.
- Exception(String message): crea un'eccezione specificando un messaggio diagnostico.
- Exception(Throwable cause): crea un'eccezione specificando la causa.
- Exception(String message, Throwable cause): crea un'eccezione specificando un messaggio diagnostico e una causa.

Ecco un esempio di eccezione personalizzata:

```
public class MyException extends Exception {

  public  MyException () {
   super();
  }
  public  MyException (String message) {
    super(message);
  }
  public Exception(String message, Throwable cause) {
    super(message, cause);
  }
  public Exception(Throwable cause) {
    super(cause);
  }
}
```

Nella maggior parte dei casi, tuttavia, è sufficiente creare una sottoclasse vuota:

```
public class MyException extends Exception {}
```

## Esempio riepilogativo

Per riassumere un argomento così importante e delicato, è opportuno studiare un esempio riepilogativo. Il seguente programma solleva una IllegalArgumentException se si cerca lanciare la riga di comando con un numero di parametri diverso da uno; quindi, apre il file specificato dall'utente e ne stampa il contenuto a video. Le operazioni di apertura, chiusura e lettura del file possono generare eccezioni: alcune di queste vengono gestite direttamente dal programma, mentre altre vengono inoltrate dal metodo main al runtime Java.

```
import java.io.*;

public class ProvaEccezioni {

  public static void main(String argv[]) throws IOException {
    if(argv.length!=1)
      throw new IllegalArgumentException("Uso: java ProvaEccezioni <filename>");

    BufferedReader i = null;
    try {
      i = new BufferedReader(new FileReader(argv[0])); // throws FileNotFoundException
      String s = i.readLine();                         // throws IOException
      while(s != null) {
        System.out.println(s);
        s = i.readLine();                              // throws IOException
      }
    }
    catch(FileNotFoundException fnfe) {
      System.out.println("Il file indicato non è stato trovato");
      fnfe.printStackTrace();
    }
    catch(IOException ioe) {
      System.out.println("Errore in lettura del file");
      ioe.printStackTrace();
    }
    finally {
      i.close();                                       // throws IOException
    }
  }
}
```

Capitolo 9

# Assert in Java: tecniche e filosofia d'uso

Andrea Gini

Una delle novità più importanti del JDK 1.4 rispetto alle precedenti versioni è l'introduzione di un servizio di assert. Si tratta della più significativa modifica introdotta nel linguaggio dal 1997, data in cui vennero aggiunte le classi interne. Questa innovazione, tuttavia, si distingue dalla precedente per almeno due ragioni: per prima cosa, l'introduzione delle assert ha comportato significative modifiche alla JVM, cosa che rende il bytecode prodotto dal nuovo compilatore incompatibile con le JVM precedenti; in secondo luogo, con le assert viene introdotto per la prima volta nel linguaggio Java un costrutto di meta programmazione tipico della programmazione logica. Invece di ordinare al computer "cosa deve fare" in un determinato momento, una assert indica piuttosto "una condizione che non si dovrebbe dovrebbe mai verificare" in un certo punto del programma nel corso dell'esecuzione. L'esistenza di un simile costrutto permette di mettere in pratica una forma semplificata di programmazione per contratto, una tecnica di sviluppo che aiuta a realizzare programmi più robusti imponendo la dichiarazione di precondizioni, postcondizioni e invarianti.

## Cosa sono le assert

Le assert sono istruzioni che controllano la verità di una condizione booleana, e provocano la chiusura del programma (mediante il lancio di un AssertionError) nel caso in cui tale condizione risulti falsa. Per esempio l'istruzione:

```
assert a + b > 0;
```

provocherà la chiusura del programma qualora la somma dei valori `a` e `b` sia uguale o inferiore a zero. In prima istanza, la semantica dell'assert si riconduce alla forma compatta di un'espressioni del tipo:

```
if( !(a+b>0))
  throw new AssertException();
```

Ma al di là dell'eleganza di un costrutto compatto, esistono sostanziali differenze tra i due casi, sia sul piano tecnico sia su quello filosofico. Da un punto di vista tecnico, il costrutto delle assert prevede la possibilità di disabilitare in blocco il controllo delle condizioni: come verrà mostrato più avanti, le assert vengono usate essenzialmente in fase di test e debugging: durante la normale esecuzione è possibile disabilitarle, eliminando in tal modo l'overhead legato alla loro gestione. Ma esiste anche una differenza assai sottile sul piano filosofico, che rende l'assert qualcosa di completamente diverso da qualsiasi altro costrutto presente in Java. Contrariamente a quanto avviene con i costrutti standard dei linguaggi imperativi, una assert non rappresenta un ordine ma una ipotesi: l'ipotesi che una certa condizione booleana sia vera in una determinata fase dell'esecuzione di un programma. La violazione di una assert causa la chiusura del programma, dal momento che si è verificato qualcosa che il programmatore non aveva previsto. Mai e in nessun caso una assert dovrà contenere direttive che influenzino la normale esecuzione del programma.

L'utilizzo delle assert permette al programmatore di verificare la consistenza interna di un programma al fine di renderlo più stabile. Nel contempo, le assert aggiungono espressività al codice, poiché formalizzano alcune ipotesi del programmatore circa lo stato del programma durante l'esecuzione, ipotesi che possono rivelarsi false a causa di un bug.

Per comprendere a fondo la filosofia di utilizzo delle assert, può essere utile una breve introduzione.

## Sherlock Holmes e la filosofia delle Assert

Nel racconto "L'avventura degli omini danzanti" Sherlock Holmes, il più famoso detective letterario, si trova a dover affrontare un caso di omicidio, per il quale viene accusata la persona sbagliata. L'apparenza a volte inganna, ma la logica, se usata nel giusto modo, può aiutare a rimettere le cose a posto:

*Lo studio era un locale non molto grande, coperto su tre pareti dai libri, con uno scrittoio di fronte ad una finestra che dava sul giardino. Per prima cosa, dedicammo la nostra attenzione al corpo del povero gentiluomo, la cui massiccia figura giaceva in mezzo alla stanza. Le vesti in disordine indicavano che era stato bruscamente risvegliato dal sonno. Il proiettile, sparato dal davanti, era rimasto nel corpo dopo avere attraversato il cuore. Non c'erano tracce di polvere da sparo, né sulla vestaglia né sulle mani. Secondo il medico, la signora invece mostrava tracce di polvere sul viso ma non sulle mani.*

*"L'assenza di tracce di polvere sulle mani non significa nulla; avrebbe avuto molto significato, invece, la loro presenza", disse Holmes. "Se il proiettile non è difettoso e la polvere non schizza*

*indietro, si possono sparare molti colpi senza che ne rimanga traccia. Ora suggerirei di rimuovere il corpo del signor Cubitt. Immagino, dottore, che lei non abbia recuperato il proiettile che ha ferito la signora?"*

*"Prima di poterlo recuperare occorre un complicato intervento chirurgico. Ma nella pistola ci sono ancora quattro proiettili. Due sono stati sparati, provocando due ferite, quindi il conto dei proiettili torna."*

*"Così sembrerebbe", convenne Holmes. "Forse può anche dirmi che fine ha fatto il proiettile che ha ovviamente colpito il bordo della finestra?"*

*Si era improvvisamente girato indicando, col lungo indice sottile, un foro che attraversava l'estremità inferiore del telaio della finestra, circa un pollice sopra il bordo.*

*"Per Giove!", esclamò l'ispettore. "Come diamine ha fatto a vederlo?"*

*"L'ho visto perché lo stavo cercando."*

*"Fantastico!", disse il dottore. "Lei ha senz'altro ragione, signore; e allora, ci deve essere stata una terza persona. Ma chi poteva essere, e come ha fatto ad andarsene?"*

*"È questo il problema che dobbiamo risolvere", disse Holmes. "Ispettore Martin, lei ricorderà che le domestiche hanno dichiarato di aver sentito odore di polvere da sparo uscendo dalle loro stanze, e che le ho detto che si trattava di un elemento di estrema importanza?"*

*"Sì, lo ricordo; ma confesso di non aver capito il motivo della sua raccomandazione."*

*"Ci porta a desumere che, al momento dello sparo, tanto la finestra che la porta della stanza erano aperte. Altrimenti, il fumo dell'esplosione non si sarebbe potuto diffondere così rapidamente nella casa. Per questo, bisognava che nella stanza ci fosse corrente. Porta e finestra, però, sono rimaste aperte solo per pochi minuti."*

*"Come può provarlo?"*

*"Perché la candela non aveva sgocciolato."*

*"Magnifico!", esclamò l'ispettore. "Magnifico!"*

*"Essendo certo che, al momento della tragedia, la finestra era aperta, ho pensato che nella faccenda poteva essere coinvolta una terza persona, che aveva sparato dall'esterno. Un proiettile diretto contro questa persona avrebbe potuto colpire il telaio della finestra. Ho cercato e, voilà, c'era il segno del proiettile."*

La tecnica investigativa di Holmes è basata sulla deduzione: la sua massima più celebre è "Se escludi l'impossibile, ciò che rimane, per quanto improbabile, non può che essere la verità". Tuttavia in informatica non è permesso escludere l'impossibile: non esiste la possibilità di realizzare programmi che dimostrino la correttezza logica di un generico altro programma.

Quando Holmes afferma "L'assenza di tracce di polvere sulle mani non significa nulla; avrebbe avuto molto significato, invece, la loro presenza", egli enuncia una verità profonda: la logica deduttiva funziona soltanto in presenza di fatti di cui sia nota la condizione di verità. In assenza di fatti non è possibile dire con certezza se una determinata ipotesi sia vera o falsa. Ciò si accorda perfettamente con il pensiero del celebre scienziato informatico E. W. Dijkstra, che era solito sostenere che "il collaudo permette di dimostrare la presenza di errori in un programma, non la loro assenza".

Se è vero che non è possibile garantire l'assenza di errori, è comunque possibile seminare alcune "trappole" nei punti critici del codice, in modo da ottenere il maggior numero possibile

di indizi qualora si verifichi un errore inaspettato. Per semplificare questo lavoro è possibile ricorrere alle assert. Fondamentalmente, le assert vengono usate per controllare tre tipi di condizione: precondizioni, postcondizioni ed invarianti. Le precondizioni sono clausole che devono risultare vere prima che venga eseguita una determinata sequenza di operazioni; le postcondizioni al contrario devono essere vere alla fine della sequenza; gli invarianti infine sono condizioni che devono sempre risultare vere.

Si osservi come Holmes riesca a falsificare un'ipotesi dapprima verificando la violazione di una condizione invariante (il numero dei proiettili sparati è superiore a quello dei proiettili di una sola pistola), quindi controllando la verità di una precondizione (la finestra aperta) unitamente alla falsità di una postcondizione (l'assenza di sgocciolamento della candela). Grazie alle assert il programmatore, al pari di Holmes, può raccogliere indizi importanti, indizi che possono aiutare a verificare le proprie ipotesi grazie alla deduzione logica.

## Sintassi delle assert

L'istruzione assert prevede due costrutti:

```
assert booleanExpression;
assert booleanExpression : message;
```

Il primo permette di specificare la condizione da controllare; il secondo contiene anche un messaggio da visualizzare in caso di violazione. Tale messaggio può contenere anche informazioni dettagliate sul caso che ha provocato il fallimento del programma, per esempio:

```
assert a + b > c : “La somma di “ + a + “ con “ + b + “ ha dato un risultato minore o uguale a “ + c;
```

## Compilazione ed esecuzione di codice con assert

L'uso delle assert richiede opzioni speciali in fase di compilazione e di esecuzione. Un esempio passo-passo dovrebbe aiutare a chiarire tutto quello che è necessario sapere. Si suggerisce quindi di copiare il seguente programma in un file dal nome AssertTest.java:

```
public class AssertTest  {
  public static void main(String args[]) {
    byte b = 0;
    for ( int i = 0; i <= 64; i++) {
      assert i >= 0;                              // precondizione
      b = (byte)(i * 2);                          // assegnamento
      assert b >= 0 : “Valore inaspettato: b = “ + b;    // postcondizione
      System.out.println(“b = “ + b);
    }
  }
}
```

La assert in quinta riga è un esempio di precondizione: essa attesta che la variabile i deve avere un valore positivo prima che venga eseguita l'operazione presente nella riga successiva. Dopo l'assegnamento in sesta riga, si può osservare invece un esempio di postcondizione, la quale asserisce che al termine dell'operazione il valore di b deve essere a sua volta positivo, dal momento che tale variabile contiene il valore di un numero positivo moltiplicato per 2. D'altra parte, il ciclo for in quarta riga contiene un errore logico: la condizione i<=64 farà in modo che l'operazione di casting presente nell'assegnamento produca, nel corso dell'ultima iterazione, un valore negativo, dal momento che una variabile di tipo byte non è in grado di contenere un valore superiore a 127. Si noti come un simile difetto logico possa nascere da un semplice errore di battitura.

Per fare in modo che il compilatore accetti il codice contenente assert, è necessario utilizzare lo speciale flag -source 1.4 da riga di comando. Tale soluzione è resa necessaria dall'esigenza di garantire la compatibilità con il codice precedente al JDK 1.4, che permetteva di usare la parola "assert" come un normale identificatore per nomi di variabile o di metodo:

```
javac -source 1.4 AssertTest.java
```

Per default le assert sono disabilitate; pertanto, se si prova a lanciare il programma con il comando:

```
java AssertTest
```

si ottiene il seguente output, che presenta come ultimo valore un numero negativo, ossia un errore dovuto all'overflow indotto dal ciclo for:

```
....
b = 122
b = 124
b = 126
b = -128
```

Per abilitare il controllo delle assert in fase di esecuzione, è necessario utilizzare il flag -ea:

```
java -ea AssertTest
```

In questo caso il programma terminerà prima di raggiungere l'overflow, e segnalerà la violazione della postcondizione:

```
....
b = 122
b = 124
b = 126
java.lang.AssertionError: Valore inaspettato: b = -128
        at AssertTest.main(AssertTest.java:7)
Exception in thread "main"
```

Come si può vedere, non è difficile utilizzare il costrutto assert nei propri programmi. I flag di attivazione dispongono comunque di una serie di opzioni avanzate, che verranno descritte nel prossimo paragrafo.

## Abilitazione e disabilitazione selettiva

Oltre al flag `-ea` (Enable Assertion) è disponibile un flag complementare `-da` (Disable Assertion). Per entrambi è possibile specificare un parametro che può assumere i seguenti valori:

- *Nessun valore*: le assert vengono abilitate o disabilitate in tutte le classi (escluse le classi di sistema, disabilitate per default).
- *NomeDiPackage*: le assert vengono abilitate o disabilitate nel package indicato e in tutti i suoi sotto package.
- ... : abilita o disabilita le assert nel package di default.
- *NomeDiClasse*: Abilita o disabilita le assert nella classe specificata.

Grazie a questi flag è possibile specificare in modo preciso e dettagliato la modalità di esecuzione. Per esempio, la seguente riga di comando esegue la classe `AssertionTest`, dopo aver abilitato le assert nel package `it.mokabyte.provaAssertion` e nei suoi eventuali sotto package:

```
java -ea:it.mokabyte.provaAssertion... AssertionTest
```

È possibile replicare ciascuno di questi flag, in modo da ottenere il risultato desiderato. Il seguente comando lancia la classe `AssertionTest` dopo aver abilitato le assert nel package `it.mokabyte.provaAssertion` e nei suoi eventuali sotto package, con l'esclusione del sottopackage `subPackage1` e della classe `Class1A`:

```
java -ea:it.mokabyte.provaAssertion... -da:it.mokabyte.provaAssertion.subPackage1...
  -da:it.mokabyte.provaAssertion.Class1A AssertionTest
```

I flag `-ea` e `-da` permettono di abilitare o disabilitare le assert su qualsiasi package, compresi i package di sistema. Tuttavia, quando si usano i flag senza parametro, le assert sono disabilitate sulle classi di sistema. Per gestire in modo esplicito l'abilitazione o la disabilitazione delle assert nelle classi di sistema, è possibile ricorrere ai flag `-esa` (Enable System Assertions) e `-dsa` (Disable System Assertions).

# Input/Output

Lorenzo Bettini

## Introduzione

In questo capitolo verrà illustrato il package java.io, che supporta il sistema fondamentale di input/output (I/O) di Java.

Nei programmi Java vengono spesso utilizzate istruzioni per stampare sullo schermo delle stringhe; utilizzare l'interfaccia a caratteri, invece che quella grafica, risulta molto comodo sia per scrivere esempi semplici, che per stampare informazioni di debug. Del resto se si scrive un'applicazione che utilizza intensamente la grafica, è comunque possibile stampare informazioni in una finestra di testo. In effetti il supporto di Java per l'I/O della console (testo) è un po' limitato, e presenta qualche complessità di utilizzo, anche nei programmi più semplici.

Comunque Java fornisce un ottimo supporto per l'I/O per quanto riguarda i file e la rete, tramite un sistema stabile e coerente. Si tratta di un ottimo esempio di libreria orientata agli oggetti che permette di sfruttare a pieno le feature della programmazione object oriented. Una volta compresi i concetti fondamentali dell'I/O di Java, è semplice sfruttare la parte restante del sistema I/O e, se si progettano le proprie classi tenendo presente la filosofia object oriented, si noterà come tali classi saranno riutilizzabili, ed indipendenti dal particolare mezzo di input/output.

## Stream

I programmi in Java comunicano (cioè effettuano l'I/O) tramite gli *stream* (in italiano *flussi*). Uno stream è un'astrazione ad alto livello che produce o consuma informazioni: rappresenta una connessione a un canale di comunicazione. Uno stream quindi è collegato a un dispositivo

fisico dal sistema I/O di Java. Gli stream possono sia leggere da un canale di comunicazione che scrivere su tale canale: quindi si parla di stream di input, e stream di output.

Gli stream si comportano in modo omogeneo, indipendentemente dal dispositivo fisico con cui sono collegati (da qui il concetto di astrazione ad alto livello). Infatti le stesse classi e gli stessi metodi di I/O possono essere applicati a qualunque dispositivo. Uno stream (astratto) di input può essere utilizzato per leggere da un file su disco, da tastiera, o dalla rete; allo stesso modo uno stream di output può fare riferimento alla console (e quindi scrivere sullo standard output), a un file (e quindi scrivere e aggiornare un file), o ancora ad una connessione di rete (e quindi spedire dei dati in rete).

Un flusso quindi rappresenta un'estremità di un canale di comunicazione a un senso solo. Le classi di stream forniscono metodi per leggere da un canale o per scrivere su un canale. Quindi un *output stream* scrive dei dati su un canale di comunicazione, mentre un *input stream* legge dati da un canale di comunicazione. Non esistono delle classi di stream che forniscano funzioni sia per leggere che per scrivere su un canale. Se si desidera sia leggere che scrivere su uno stesso canale di comunicazione si dovranno aprire due stream (uno di input ed uno di output) collegati allo stesso canale.

Di solito un canale di comunicazione collega uno stream di output al corrispondente stream di input. Tutti i dati scritti sullo stream di output, potranno essere riletti (nello stesso ordine) dallo stream di input. Poiché, come si è già detto, gli stream sono indipendenti dal particolare canale di comunicazione, essi mettono a disposizione uno strumento semplice e uniforme per la comunicazione fra applicazioni. Due applicazioni che si trovano su due macchine diverse, ad esempio, potrebbero scambiarsi i dati tramite uno stream collegato alla rete, oppure un'applicazione può semplicemente comunicare con l'utente utilizzando gli stream collegati alla console. Gli stream implementano una struttura FIFO (*First In First Out*), nel senso che il primo dato che sarà scritto su uno stream di output sarà il primo che verrà letto dal corrispondente stream di input. Fondamentalmente, quindi, gli stream mettono a disposizione un accesso sequenziale alle informazioni scambiate.

Quando si parla di input/output, si parla anche del problema dell'azione bloccante di una richiesta di input (il concetto di input/output tra l'altro si ritrova anche nelle architetture dei processori). Ad esempio, se un thread cerca di leggere dei dati da uno stream di input che non contiene dati, verrà bloccato finché non saranno presenti dei dati disponibili per essere letti. In effetti, quando un thread cerca di leggere dei caratteri immessi da un utente da tastiera, rimarrà in attesa finché l'utente non inizierà a digitare qualcosa. Il problema dell'azione bloccante è valido anche per le operazioni di output: se si cerca di scrivere qualcosa in rete, si rimarrà bloccati finché l'operazione non sarà terminata. Questo può avvenire anche quando si scrive su un file su disco, ma le operazioni in rete di solito sono le più lente.

Il thread bloccato sarà risvegliato solo quando sarà stata completata l'operazione bloccante. Se si vuole evitare di essere bloccati da queste operazioni si dovrà utilizzare il multithreading; si vedranno degli esempi nel capitolo che riguarda il networking.

## Le classi

Le classi degli stream sono contenute nel pacchetto `java.io`, che dovrà quindi essere incluso nei programmi che ne fanno uso.

Tutti gli stream fanno parte di una gerarchia. In realtà si hanno due sottogerarchie: una per gli stream di output ed una per quella di input.

In cima a questa gerarchia ci sono due classi astratte i cui nomi sono abbastanza ovvi: InputStream e OutputStream. Trattandosi di classi astratte, non si potranno istanziare direttamente oggetti appartenenti a queste classi. Comunque si possono dichiarare delle variabili appartenenti a queste classi (per i programmatori C++, si ricorda che le variabili dichiarate sono in effetti dei riferimenti o puntatori, e quindi dichiarando una variabile non si istanzia automaticamente un oggetto di tale classe), e a queste si potrà assegnare qualsiasi oggetto appartenente ad una classe derivata (l'analogia con il C++ prosegue: un puntatore a una classe base può puntare a un qualsiasi oggetto appartenente a una classe derivata); in questo modo si potrà utilizzare a pieno il polimorfismo, rendendo le proprie classi indipendenti dal particolare stream (e quindi anche dal particolare canale di comunicazione).

Java ovviamente mette a disposizione diverse sottoclassi che specializzano gli stream per i diversi dispositivi e canali di comunicazione, ma vediamo prima i metodi messi a disposizione da queste due classi base.

# La classe OutputStream

La classe OutputStream rappresenta una porta verso un canale di comunicazione; tramite questa porta si possono scrivere dati sul canale con il quale la porta è collegata. Si ricorda che si tratta di una classe astratta, che quindi fornisce un'interfaccia coi metodi caratteristici di ogni stream di output. Saranno le sottoclassi a fornire un'implementazione effettiva di tali metodi, che ovviamente dipenderà dal particolare canale di comunicazione.

## Descrizione classe

```
public abstract class OutputStream extends Object
```

Trattandosi di una classe astratta, non sono presenti costruttori utilizzabili direttamente.

## Metodi

```
public abstract void write(int b) throws IOException
```

Viene accettato un singolo byte, che verrà scritto sul canale di comunicazione con il quale lo stream è collegato. Notare che, nonostante l'argomento sia di tipo intero, verrà scritto solo il byte meno significativo. Ovviamente si tratta di un metodo astratto, in quanto la scrittura dipende fortemente dal particolare dispositivo fisico del canale di comunicazione.

```
public void write(byte b[], int off, int len) throws IOException
```

```
public void write(byte b[]) throws IOException
```

Questi metodi permettono di scrivere un array di byte sul canale di comunicazione. È possibile scrivere l'intero array (secondo metodo), o solo una parte (primo metodo), specificando l'indice del primo elemento (off), e il numero di elementi (len). Il secondo metodo, nell'implementazione di default, richiama semplicemente il primo sull'intero array. A sua volta il primo metodo, nella sua implementazione di default, richiama il numero di volte necessario il metodo write(int b). Il metodo bloccherà il chiamante fino a che tutti i byte dell'array non saranno stati scritti.

```
public void flush() throws IOException
```

Questo metodo effettua il *flush* dei dati bufferizzati nello stream, cioè fa in modo che eventuali dati non ancora scritti effettivamente, vengano scritti nel canale di comunicazione. A volte infatti, per motivi di ottimizzazione e performance, i dati scritti nello stream non vengono scritti immediatamente nel canale di comunicazione, ma vengono tenuti temporaneamente in un buffer. Con questo metodo si fa in modo che i dati presenti nel buffer vengano scritti effettivamente sul canale. Quando si tratta di comunicazioni in rete, la tecnica della "bufferizzazione" è quasi d'obbligo, per ovviare alla lentezza di tali comunicazioni.

```
public void close() throws IOException
```

Con questo metodo si chiude lo stream e quindi il canale di comunicazione. Prima della chiusura tutti i dati eventualmente bufferizzati vengono sottoposti a flush; questo può comportare il dover attendere (e quindi rimanere bloccati) fino al completamente dell'operazione di scrittura.

L'eccezione IOException può essere lanciata per vari motivi che riguardano dei problemi del canale di comunicazione. Il tipo esatto dell'eccezione dipende quindi dal particolare canale. Tipicamente le operazioni sugli stream dovrebbero essere racchiuse nei classici blocchi try–catch–finally, o fare in modo che il metodo che li utilizza dichiari di lanciare una tale eccezione.

# La classe InputStream

La classe InputStream è la classe complementare della classe OutputStream, che fornisce funzionalità per l'input, quindi per la lettura di dati da un canale di comunicazione. Quanto si è detto sui metodi astratti è valido anche per questa classe.

## Descrizione classe

```
public abstract class InputStream extends Object
```

## Metodi

Questa classe fornisce metodi per leggere byte, per determinare il numero di byte disponibili per essere letti senza rimanere bloccati, e per saltare o rileggere dei dati. Come è già stato accennato, leggere da uno stream che non contiene dati bloccherà il thread che ha effettuato

l'operazione di lettura. Se alcuni dati sono già arrivati dal canale di comunicazione, verranno messi temporaneamente in un buffer in attesa di essere effettivamente letti. Quando, a questo punto, un thread cercherà di leggere dallo stream, lo potrà fare immediatamente senza bisogno di attendere e di bloccarsi.

```
public abstract int read() throws IOException
```

Questo metodo legge un singolo byte, aspettando eventualmente che ve ne sia uno disponibile. Ancora una volta, pur trattandosi di un `int`, il valore restituito sarà comunque compreso fra `0` e `255`. Se viene raggiunta la fine dello stream, verrà restituito il valore `-1`. Il concetto di fine dello stream dipende dal particolare canale di comunicazione che si sta utilizzando (ad esempio nel caso di un file rappresenta la fine del file). Si tratta di un metodo astratto perché la lettura di dati da uno stream dipende dal particolare canale di comunicazione con cui lo stream è collegato.

```
public int read(byte b[], int off, int len) throws IOException
```

```
public int read(byte b[]) throws IOException
```

Con questi metodi è possibile leggere una serie di byte e memorizzarli nell'array specificato. È possibile specificare anche il numero di byte da leggere (`len`) e memorizzare nell'array, specificando l'indice iniziale (`off`). L'array dovrà già essere stato allocato. Si tratta ovviamente di un metodo bloccante, se non sono presenti dati da leggere. Il metodo restituisce inoltre il numero di byte letti. Infatti non è detto che venga letto esattamente il numero di byte richiesti: vengono letti i dati che possono essere letti immediatamente senza necessità di attendere, e questi possono essere in numero inferiore a quelli effettivamente richiesti. L'implementazione di default del secondo metodo è quella di richiamare il primo su tutto l'array. A sua volta l'implementazione di default del primo è di richiamare ripetutamente il metodo `read()`.

```
public abstract int available() throws IOException
```

Restituisce il numero di byte che sono disponibili nello stream per essere letti senza attendere.

```
public void close() throws IOException
```

Chiude lo stream e il canale di comunicazione con cui lo stream è collegato. I dati non ancora letti andranno persi.

```
public long skip(long n) throws IOException
```

Vengono saltati e scartati `n` byte presenti nello stream. Questo è utile se si vogliono ignorare dei byte, ed è più efficiente che leggere i byte e ignorarli. Il metodo restituisce il numero di byte effettivamente saltati; questo perché, per vari motivi, può non essere possibile saltare esattamente il numero di byte richiesto.

```
public synchronized void mark(int readlimit)

public synchronized void reset() throws IOException
```

Marca la posizione corrente all'interno dello stream. Una successiva chiamata al metodo reset riposizionerà lo stream alla precedente posizione marcata. Dopo la chiamata del metodo reset letture successive leggeranno dall'ultima posizione marcata. Con il parametro readlimit si specifica il numero massimo di byte che saranno letti, prima che la posizione marcata non sia più valida. Se sono letti più di readlimit byte, una successiva chiamata di reset potrebbe fallire.

Questi due metodi possono risultare utili nelle situazioni in cui vi sia bisogno di leggere alcuni byte prima di capire quale tipo di dati è presente nello stream. Se si deve decodificare tali dati, e si hanno vari tipi di decodificatori, quando un decodificatore si rende conto che non sono dati che lo riguardano, può "rimettere a posto" i dati già letti, rendendoli disponibili ad un altro decodificatore.

```
public boolean markSupported()
```

Permette di capire se lo stream corrente gestisce il corretto funzionamento delle operazioni di mark e reset.

Anche nel caso di InputStream l'eccezione IOException può essere lanciata in varie occasioni.

# Gli stream predefiniti

Il pacchetto java.lang, incluso automaticamente da tutti i programmi Java, definisce alcuni stream predefiniti, contenuti nella classe System. Si tratta di tre variabili statiche e pubbliche (quindi utilizzabili in qualunque parte del programma, senza aver istanziato un oggetto System) denominate in, out e err. Queste si riferiscono rispettivamente allo standard input, che per default è la tastiera, al flusso standard di output, che per default è lo schermo, e al flusso standard di errori che, anche in questo caso, per default è lo schermo. Tali stream possono essere reindirizzati quando si lancia il programma da linea di comando utilizzando > e < (per questo si rimanda al sistema operativo che si utilizza).

## Esempi

Si prenderanno ora in considerazione due semplici esempi che utilizzano tali stream predefiniti:

```
import java.io.*;

public class OutSample {
 public static void main (String args[]) throws IOException {
    for (int i = 0; i < args.length; ++ i) {
       synchronized(System.out) {
```

```
          for (int j = 0; j < args[i].length (); ++j)
            System.out.write ((byte) args[i].charAt (j));
          System.out.write ('\n');   // scrive un invio
          System.out.flush ();    // scarica il buffer
        }
      }
  }
}
```

Questo semplice programma scrive sullo schermo i vari argomenti passati sulla linea di comando. Viene utilizzato il metodo write per scrivere un byte alla volta, ed il metodo flush per essere sicuri che ogni stringa passata venga stampata subito. Si può notare che il metodo main dichiara di poter lanciare un'eccezione IOException; in effetti i metodi write e flush possono lanciare tali eccezioni.

Un po' meno chiaro può risultare l'utilizzo di un blocco sincronizzato. In questo caso non sarebbe necessario in quanto non si usano più thread. Nel caso di un programma con più thread è bene sincronizzare l'accesso alla variabile System.out in modo che, quando un thread ha iniziato a scrivere su tale stream, non venga interrotto prima che abbia finito; nello stream altrimenti sarebbero presenti informazioni rovinate e mischiate.

Un'alternativa potrebbe essere quella di scrivere una stringa alla volta, invece dei suoi singoli byte. Per far questo si deve convertire la stringa in un array di byte, e poi richiamare il metodo write appropriato. Vale a dire che al posto del ciclo for più interno si sarebbe potuto scrivere

```
byte buffer[] = new byte[args[i].length()];
msg.getBytes (0, args[i].length (), buffer, 0);
System.out.write (buffer);
```

In effetti la variabile out appartiene alla classe PrintStream, che specializza un OutputStream per scrivere dati in formato testo (e quindi adatto per scrivere dati sullo schermo). Questa classe mette a disposizione due metodi molto utilizzati per stampare facilmente stringhe e altri dati come testo: si tratta dei metodi print e println (quest'ultimo si distingue dal precedente perché aggiunge un newline dopo la stampa). In effetti il programma precedente può essere riscritto in modo molto più semplice:

```
public class OutSamplePrint {
 public static void main (String args[]) throws IOException {
   for (int i = 0; i < args.length; ++ i)
     System.out.println(i + ": " + args[i]);
 }
}
```

Come si vede non c'è bisogno di tradurre la stringa in un array di byte, in quanto i metodi suddetti gestiscono direttamente le stringhe, e sono anche in grado di tradurre dati di altro tipo (ad esempio i è un intero) in stringa (infatti questo programma stampa le stringhe immesse da riga di comando insieme alla numerazione). Non c'è nemmeno bisogno di sincronizzarsi su System.out in quanto questi metodi sono già dichiarati come sincronizzati.

Ecco adesso un semplice programma che legge i caratteri immessi da tastiera e li ristampa sullo schermo. In questo caso si utilizzerà anche la variabile System.in.

```
import java.io.*;

public class InSample {
 static public void main (String args[]) throws IOException {
    int c;
    while ((c = System.in.read ()) >= 0)
       System.out.print((char)c);
 }
}
```

Come si vede, dei caratteri da tastiera vengono letti e poi stampati sullo schermo (la conversione esplicita a char è necessaria, altrimenti verrebbe stampato un numero). Da notare che il metodo read memorizza in un buffer i caratteri digitati e li restituisce solo quando l'utente preme Invio. Chiaramente questo metodo non è molto indicato per un input interattivo da console.

Ancora una volta, può essere più efficiente utilizzare dei buffer per ottimizzare le prestazioni. Per far questo basta cambiare il corpo del main con il seguente:

```
    byte buffer[] = new byte[8];
    int numberRead;
    while ((numberRead = System.in.read (buffer)) > -1)
       System.out.write (buffer, 0, numberRead);
```

Questo semplice programma può essere utilizzato anche per visualizzare il contenuto di un file di testo: basterà semplicemente ridirezionare lo standard input (la tastiera) su un file. Ad esempio con il seguente comando

```
java InSample < InSample.java
```

si visualizzerà sullo schermo il contenuto del sorgente del programma stesso.

Si vedranno adesso alcune classi che specializzano gli stream di input e output. Come si è visto le classi base offrono solo metodi per scrivere singoli byte e al massimo array di byte. Spesso invece si ha la necessità di leggere e/o scrivere stringhe o numeri, quindi si avrebbe bisogno di stream che forniscano metodi per effettuare direttamente queste operazioni, senza dover manualmente effettuare conversioni.

## Stream filtro

Si vedrà ora il concetto di stream filtro (*filter stream*), cioè uno stream che fornisce metodi ad alto livello per inviare o ricevere i dati primitivi di Java su un qualsiasi stream di comunicazione.

Uno stream filtro agisce appunto come un filtro per uno stream già esistente, aggiungendo

funzionalità ad alto livello. Tra l'altro questo permette di tralasciare tutti i dettagli su come i dati vengono memorizzati in uno stream (ad esempio se un intero viene memorizzato partendo dal byte più alto o da quello più basso).

È necessario quindi fornire uno stream già esistente ad uno stream filtro. Ad uno stream filtro di input passeremo uno stream di input qualsiasi (in pratica un oggetto di classe InputStream), così come ad uno stream filtro di output passeremo uno stream di output qualsiasi (un oggetto di classe OutputStream). Anche gli stream filtro sono sottoclassi delle classi base InputStream e OutputStream, quindi è possibile costruire una serie di stream filtro in cascata, a seconda delle varie esigenze. Ci sarà modo di vedere alcuni esempi successivamente.

## Le classi FilterOutputStream e FilterInputStream

Queste sono le classi base per ogni stream filtro, e non sono altro che template (modelli) per tutti gli altri stream filtro. L'unica funzionalità aggiuntiva che mettono a disposizione è il fatto di poter passare ai loro costruttori un qualsiasi stream con il quale collegarsi (quindi rispettivamente un OutputStream e un InputStream). Gli unici metodi che mettono a disposizione sono gli stessi che sono presenti nella classe base. La semplice azione di default sarà quella di richiamare il metodo corrispondente dello stream con il quale sono collegati. La loro utilità si riduce quindi a fornire un'interfaccia uniforme per tutti gli altri stream filtro, e ovviamente a fornire una classe base comune.

## Le classi DataOutputStream e DataInputStream

Queste classi sono fra le più utilizzate in quanto mettono a disposizione proprio le funzionalità che cercavamo negli stream filtro: forniscono metodi rispettivamente per scrivere e leggere tutti i tipi primitivi del linguaggio (stringhe, interi, ecc.).

Ovviamente questi due stream, come spesso accade negli stream filtro, devono lavorare in coppia affinché le comunicazioni di informazioni abbiano successo: se da una parte si utilizza un DataOutputStream per spedire una stringa, dall'altra parte ci dovrà essere in ascolto un DataInputStream, che sia in grado di decodificare la stringa ricevuta dal canale di comunicazione. Infatti i metodi di questi stream filtro si occupano, rispettivamente, di codificare e decodificare i vari tipi di dato. Non sarà necessario preoccuparsi dell'ordine dei byte di un intero o della codifica di una stringa, ovviamente purché tali stream siano utilizzati in coppia.

Nonostante non ci si debba preoccupare della codifica dei dati spediti, può comunque essere interessante sapere che questi stream utilizzano il *network byte order* per la memorizzazione dei dati: il byte più significativo viene scritto per primo (e dall'altra parte letto per primo). In questo modo le applicazioni scritte in Java, potranno comunicare dati con questi stream, con qualsiasi altro programma scritto in un altro linguaggio che usi la convenzione del network byte order.

## Descrizione classe DataOutputStream

```
public class DataOutputStream
extends FilterOutputStream implements DataOutput
```

L'unica cosa da notare è l'interfaccia DataOutput. Questa interfaccia, insieme alla simmetrica DataInput, descrive gli stream che scrivono e leggono (rispettivamente) dati in un formato indipendente dalla macchina.

## Costruttore

```
public DataOutputStream(OutputStream out)
```

Come già accennato quando si è parlato in generale degli stream filtro, viene passato al costruttore lo stream sul quale si agisce da filtro. Vale la pena di ricordare che si passa un OutputStream, quindi, trattandosi della classe base di tutti gli stream di output, si può passare un qualsiasi stream di output.

## Metodi

I metodi seguenti fanno parte della suddetta interfaccia DataOutput. A questi vanno aggiunti i metodi derivati dalla classe base, che non verranno descritti (il loro nome è di per sé molto esplicativo).

```
public final void writeBoolean(boolean v) throws IOException

public final void writeByte(int v) throws IOException

public final void writeShort(int v) throws IOException

public final void writeChar(int v) throws IOException

public final void writeInt(int v) throws IOException

public final void writeLong(long v) throws IOException

public final void writeFloat(float v) throws IOException

public final void writeDouble(double v) throws IOException
```

Come si può notare, esiste un metodo per ogni tipo di dato primitivo di Java. Il loro significato dovrebbe essere abbastanza immediato. I prossimi metodi invece meritano una spiegazione un po' più dettagliata.

```
public final void writeBytes(String s) throws IOException
```

Questo metodo scrive una stringa sullo stream collegato come una sequenza di byte. Viene scritto solo il byte più basso di ogni carattere, quindi può essere utilizzato per trasmettere dei dati in formato ASCII a un dispositivo come un terminale carattere, o un client scritto in C. La lunghezza della stringa non viene scritta nello stream.

```
public final void writeChars(String s) throws IOException
```

La stringa passata viene scritta come sequenza di caratteri. Ogni carattere viene scritto come una coppia di byte. Non viene scritta la lunghezza della stringa, né il terminatore.

```
public final void writeUTF(String str) throws IOException
```

La stringa viene scritta nel formato Unicode UTF-8 in modo indipendente dalla macchina. La stringa viene scritta con una codifica in modo tale che ogni carattere viene scritto come un solo byte, due byte, o tre byte. I caratteri ASCII saranno scritti come singoli byte, mentre i caratteri più rari vengono scritti con tre byte. Inoltre i primi due byte scritti rappresentano il numero di byte effettivamente scritti. Quindi la lunghezza della stringa viene scritta nello stream.

Tutti questi metodi possono lanciare l'eccezione IOException; questo perché viene usato il metodo write dello stream con il quale lo stream filtro è collegato, che può lanciare un'eccezione di questo tipo.

## Descrizione classe DataInputStream

```
public class DataInputStream extends FilterInputStream implements DataInput
```

Valgono le stesse considerazioni fatte riguardo alla classe DataOutputStream.

## Costruttore

```
public DataInputStream(InputStream in)
```

Anche in questo caso si passa un InputStream al costruttore.

## Metodi

Sono presenti i metodi simmetrici rispetto a quelli di DataOutputStream.

```
public final boolean readBoolean() throws IOException
```

```
public final byte readByte() throws IOException
```

```
public final int readUnsignedByte() throws IOException
```

```
public final short readShort() throws IOException

public final int readUnsignedShort() throws IOException

public final char readChar() throws IOException

public final int readInt() throws IOException

public final long readLong() throws IOException

public final float readFloat() throws IOException

public final double readDouble() throws IOException

public final String readUTF() throws IOException
```

Metodi che meritano particolare attenzione sono i seguenti:

```
public final void readFully(byte b[], int off, int len) throws IOException

public final void readFully(byte b[]) throws IOException
```

Questi metodi leggono un array di byte o un sottoinsieme, ma bloccano il thread corrente finché tutto l'array (o la parte di array richiesta) non viene letto. Viene lanciata un'eccezione EOFException se viene raggiunto prima l'EOF. A tal proposito si può notare che non può essere restituito il numero -1 per segnalare l'EOF, in quanto se si sta leggendo un intero, -1 è un carattere intero accettabile. Per questo motivo si ricorre all'eccezione suddetta.

```
public final static String readUTF(DataInput in) throws IOException
```

Si tratta di un metodo statico che permette di leggere una stringa con codifica UTF, dall'oggetto in, quindi un oggetto (in particolare uno stream) che implementi l'interfaccia DataInput.

Anche in questo caso può essere lanciata un'eccezione IOException. In particolare la suddetta eccezione EOFException deriva da IOException. Un'altra eccezione (sempre derivata da IOException) che può essere lanciata è UTFDataFormatException, nel caso in cui i dati ricevuti dal metodo readUTF non siano nel formato UTF.

## Classi BufferedOutputStream e BufferedInputStream

Talvolta nelle comunicazioni è molto più efficiente bufferizzare i dati spediti. Questo è senz'altro vero per le comunicazioni in rete, ma può essere vero anche quando si deve scrivere o leggere da un file (anche se a questo pensaautomaticamente il sistema operativo sottostante).

Richiamando i metodi di scrittura della classe BufferedOutputStream, i dati verranno memorizzati temporaneamente in un buffer interno (quindi in memoria), finché non viene chiamato

il metodo flush, che provvederà a scrivere effettivamente i dati nello stream con cui il filtro è collegato, oppure finché il buffer non diventa pieno.

Quindi è molto più efficiente scrivere dei dati su un canale di comunicazione utilizzando un DataOutputStream collegato a un BufferedOutputStream. Ad esempio se si utilizza un DataOutputStream collegato direttamente a un canale di comunicazione di rete, e si scrive un intero con il metodo writeInt, è molto probabile che il primo byte dell'intero scritto sarà spedito subito in rete in un pacchetto. Un altro pacchetto — o forse più pacchetti — sarà utilizzato per i rimanenti byte. Sarebbe molto più efficiente scrivere tutti i byte dell'intero in un solo pacchetto e spedire quel singolo pacchetto. Se si costruisce un DataOutputStream su un BufferedOutputStream si otterranno migliori prestazioni. Questo è, tra l'altro, un esempio di due stream filtro collegati in cascata.

Se da una parte della comunicazione c'è un BufferedOutputStream che scrive dei dati, non è detto che dall'altra parte ci debba essere un BufferedInputStream in ascolto: in effetti questi stream filtro non codificano l'output ma semplicemente effettuano una bufferizzazione.

Comunque converrebbe utilizzare anche in lettura uno stream bufferizzato, cioè un BufferedInputStream. Utilizzare un buffer in lettura significa leggere i dati dal buffer interno, e solo quando nuovi dati, non presenti nel buffer, dovranno essere letti, si accederà al canale di comunicazione.

## Descrizione classe BufferedOutputStream

```
public class BufferedOutputStream
extends FilterOutputStream
```

## Costruttori

```
public BufferedOutputStream(OutputStream out)

public BufferedOutputStream(OutputStream out, int size)
```

Nel primo caso viene creato uno stream bufferizzato collegato allo stream di output out; la dimensione del buffer sarà quella di default, cioè 512 byte. Nel secondo caso è possibile specificare la dimensione del buffer. I dati scritti in questo stream saranno scritti sullo stream collegato out solo quando il buffer è pieno, o verrà richiamato il metodo flush.

## Metodi

Come si è visto la classe deriva direttamente da FilterOutputStream e l'unico metodo che aggiunge a quelli della classe base (cioè quelli di OutputStream) è il metodo flush.

```
public synchronized void flush() throws IOException
```

Questo metodo fa sì che i dati contenuti nel buffer siano effettivamente scritti sullo stream collegato.

## Descrizione classe BufferedInputStream

```
public class BufferedInputStream extends FilterInputStream
```

## Costruttori

```
public BufferedInputStream(InputStream in)
public BufferedInputStream(InputStream in, int size)
```

Viene creato uno stream di input bufferizzato collegato allo stream si input in; è possibile anche in questo caso specificare la dimensione del buffer, o accettare la dimensione di default (512 byte).

## Metodi

Non vengono aggiunti metodi, e quindi si hanno a disposizione solo quelli di FilterInputStream (cioè solo quelli di InputStream); l'unica differenza è che tali metodi faranno uso del buffer interno.

# Stream per l'accesso alla memoria

Java supporta l'input e l'output di array di byte tramite l'uso delle classi ByteArrayOutputStream e ByteArrayInputStream. Questi stream non sono collegati con un canale di comunicazione vero e proprio: queste classi infatti utilizzano dei buffer di memoria come sorgente e come destinazione dei flussi di input e output. In questo caso, tali stream non devono essere utilizzati necessariamente insieme.

Vi sono poi le classi PipedInputStream e PipedOutputStream, che permettono la comunicazione, tramite appunto memoria, di due thread di un'applicazione. Un thread leggerà da un lato della *pipe* e riceverà tutto quello che sarà scritto dall'altro lato da altri thread. Questi stream saranno creati sempre in coppia; un lato della *pipe* viene creato senza essere connesso, mentre l'altro sarà creato connettendolo con il primo. Quindi basta collegare uno stream con l'altro, e non entrambi.

## Descrizione classe ByteArrayInputStream

```
public class ByteArrayInputStream extends InputStream
```

Questa classe crea uno stream di input da un buffer di memoria, in particolare da un array di byte.

## Costruttori

```
public ByteArrayInputStream(byte buf[])
public ByteArrayInputStream(byte buf[], int offset, int length)
```

Con questi costruttori si può specificare l'array (o una parte dell'array nel secondo caso) con il quale lo stream sarà collegato.

## Metodi

Tale classe non mette a disposizione nuovi metodi, semplicemente ridefinisce i metodi della classe base InputStream. In particolare chiamando il metodo read, in una delle sue forme, verranno letti i byte dell'array collegato, fino a che non sarà raggiunta la fine dell'array, e in tal caso sarà restituito EOF. Inoltre la semantica del metodo reset è leggermente differente: resettare un ByteArrayInputStream vuol dire ripartire sempre dall'inizio dell'array, in quanto il metodo mark marca sempre la posizione iniziale.

## Descrizione classe ByteArrayOutputStream

```
public class ByteArrayOutputStream extends OutputStream
```

Questa classe crea uno stream di output su un array di byte, ed è un po' più potente della sua classe complementare: permette all'array di byte con il quale è collegata di crescere dinamicamente, man mano che vengono aggiunti nuovi dati. Il buffer di memoria può essere estratto e utilizzato.

## Costruttori

```
public ByteArrayOutputStream()
public ByteArrayOutputStream(int size)
```

È possibile specificare la dimensione iniziale del buffer o accettare quella di default (32 byte).

## Metodi

Anche in questo caso il metodo reset() acquista un significato particolare: svuota il buffer, e successive scritture memorizzeranno i dati a partire dall'inizio. Vi sono poi alcuni metodi aggiunti:

```
public int size()
```

Viene restituito il numero di byte che sono stati scritti nel buffer (da non confondersi con la dimensione del buffer, che può essere anche maggiore).

```
public synchronized byte[] toByteArray()
```

Viene restituito un array di byte rappresentante una copia dei dati scritti nel buffer. Il buffer interno non sarà resettato da questo metodo, quindi successive scritture nello stream continueranno a estendere il buffer.

```
public String toString()
```

Viene restituita una stringa rappresentante una copia del buffer dello stream. Anche in questo caso il buffer dello stream non viene resettato. Ogni carattere della stringa corrisponderà al relativo byte del buffer.

```
public synchronized void writeTo(OutputStream out) throws IOException
```

I contenuti del buffer dello stream vengono scritti nello stream di output `out`. Anche in questo caso il buffer dello stream non viene resettato. Se si verificano degli errori durante la scrittura nello stream di output `out` verrà sollevata un'eccezione `IOException`.

Ecco adesso un piccolo esempio che fa uso dei suddetti stream. Le stringhe che vengono passate sulla riga di comando vengono tutte inserite in `ByteArrayOutputStream`. Il buffer dello stream viene estratto e su tale array di byte viene costruito un `ByteArrayInputStream`. Da questo stream verranno poi estratti e stampati sullo schermo tutti i byte.

```
Import java.io.* ;

public class ByteArrayIOSample {
 public static void main(String args[]) throws IOException {
    ByteArrayOutputStream oStream = new ByteArrayOutputStream();



for (int i = 0; i < args.length; i++)
       for (int j = 0; j < args[i].length(); j++)
          oStream.write(args[i].charAt(j));

 // per la concatenazione a stringa viene
 // chiamato toString()
    System.out.println("oStream: " + oStream);

    System.out.println("size: " + oStream.size());

    ByteArrayInputStream iStream = new ByteArrayInputStream(oStream.toByteArray());

    System.out.println("Byte disponibili: " + iStream.available());
    int c ;
    while((c = iStream.read()) != -1)
       System.out.write(c);
 }
}
```

# Descrizione classe PipedOutputStream

```
public class PipedOutputStream extends OutputStream
```

## Costruttori

```
public PipedOutputStream() throws IOException

public PipedOutputStream(PipedInputStream snk) throws IOException
```

Si può creare un PipedOutputStream e poi connetterlo con un PipedInputStream, oppure lo si può passare direttamente al costruttore, se già esiste.

## Metodi

Sono disponibili i metodi standard della classe OutputStream ed in più è presente il metodo per connettere lo stream con un PipedInputStream:

```
public void connect(PipedInputStream src) throws IOException
```

Se si scrive su un PipedOutputStream, e il thread che è in ascolto sul corrispondente PipedInputStream termina, si otterrà un'IOException.

Questi stream sono implementati con un buffer di memoria, e se il buffer diventa pieno, una successiva chiamata al metodo write bloccherà il thread che scrive sullo stream, finché il thread in ascolto sullo stream di input corrispondente non legge qualche byte. Se questo thread termina, l'eccezione suddetta evita che l'altro processo rimanga bloccato indefinitamente.

## Descrizione classe PipedInputStream

Per questa classe, che è la relativa classe di lettura della precedente, valgono le stesse annotazioni fatte per la classe PipedOutputStream.

```
public class PipedInputStream extends InputStream
```

## Costruttori

```
public PipedInputStream() throws IOException

public PipedInputStream(PipedOutputStream src) throws IOException
```

## Metodi

```
public void connect(PipedOutputStream src) throws IOException
```

Anche in questo caso si deve evitare che un thread bloccato a leggere da un PipedInputStream, rimanga bloccato indefinitamente; se viene chiamato il metodo read su uno stream vuoto, verrà sollevata un'eccezione IOException. Segue un semplice esempio che illustra l'utilizzo di questi due stream per la comunicazione fra due thread (il thread principale e un thread parallelo):

```
import java.io.* ;

public class PipedIOSample extends Thread {
 protected DataInputStream iStream ;

 public PipedIOSample(InputStream i) {
      this.iStream = new DataInputStream(i);
 }

 public void run() {
    try {
      String str;
      while (true) {
         str = iStream.readUTF();
         System.out.println("Letta: " + str);
      }
    }


catch (IOException e) {
      e.printStackTrace();
    }
 }

 public static void main(String args[]) throws IOException {
    PipedOutputStream o = new PipedOutputStream();
    PipedInputStream iStream = new PipedInputStream(o);
    DataOutputStream oStream = new DataOutputStream(o);

    (new PipedIOSample(iStream)).start();

    for (int i = 0; i < args.length; i++) {
      System.out.println("Scrivo: " + args[i]);
      oStream.writeUTF(args[i]);
    }

    oStream.close();
 }
}
```

Come si può notare viene creato un PipedOutputStream senza specificare nessuno stream da collegare; poi viene creato un PipedInputStream collegato al precedente stream. A questo punto i due stream sono connessi e tutto quello che viene scritto sullo stream di output potrà essere letto da quello di input. L'idea è quella di scrivere le stringhe da passare sulla riga di comando sullo stream di output; tali stringhe saranno lette da un altro thread sullo stream (sempre di tipo piped) di input. In particolare invece di utilizzare più volte il me-

todo write per scrivere un singolo byte alla volta, utilizziamo un DataOutputStream collegato al PipedOutputStream, e scriviamo una stringa alla volta con il metodo readUTF. Allo stesso modo il thread che legge le stringhe lo farà tramite un DataInputStream collegato allo stream passato al costruttore. Vale la pena di notare che al costruttore viene passato un InputStream generico. Il thread che legge le stringhe lo fa in un ciclo infinito, che terminerà non appena verrà chiuso lo stream di output (ultima istruzione del main), a causa dell'eccezione IOException.

# I file

Trattando l'input/output non si può certo tralasciare l'argomento file. Java fornisce l'accesso ai file tramite gli stream. In questo modo, per la genericità degli stream, un'applicazione progettata per leggere e/o scrivere utilizzando le classi InputStream e OutputStream, può utilizzare i file in modo trasparente.

Java inoltre mette a disposizione altre classi per facilitare l'accesso ai file e alle directory

## Descrizione classe File

```
public class File extends Object implements Serializable
```

La classe File fornisce l'accesso a file e directory in modo indipendente dal sistema operativo. Tale classe mette a disposizione una serie di metodi per ottenere informazioni su un certo file e per modificarne gli attributi; tramite questi metodi, ad esempio, è possibile sapere se un certo file è presente in una certa directory, se è a sola lettura, e via dicendo.

Si è parlato di indipendenza dal sistema operativo: effettivamente ogni sistema operativo utilizza convenzioni diverse per separare le varie directory in un *path*. Quando si specifica un file e/o un path, si suppone che vengano utilizzate le convenzioni del sistema operativo sottostante. I vari metodi che sono messi a disposizione dalla classe permettono di ottenere le informazioni relative a tali convenzioni. Inoltre è possibile cancellare file, rinominarli, e ottenere la lista dei file contenuti in una certa directory.

## Costruttori

```
public File(String path)
public File(String path, String name)
public File(File dir, String name)
```

È possibile creare un oggetto File specificando un path e anche un nome di file. Il path deve essere specificato utilizzando le convenzioni del sistema operativo sottostante. Se viene specificato anche il nome del file, oltre al percorso, verrà creato un path concatenando il percorso specificato ed il file con il separatore utilizzato dal sistema operativo. Con la terza versione è possibile specificare la directory del file tramite un altro oggetto File.

# Metodi

Come già detto, tale classe è utile per avere un meccanismo in grado di utilizzare file e directory in modo indipendente dalle convenzioni del sistema operativo e per eseguire le classiche operazioni sui file e sulle directory. Tali metodi non lanciano un'IOException, in caso di fallimento, ma restituiscono un valore booleano.

```
public String getName()
```

```
public String getPath()
```

Restituiscono rispettivamente il nome e il percorso dell'oggetto File.

```
public String getAbsolutePath()
```

```
public String getCanonicalPath() throws IOException
```

Restituiscono rispettivamente il percorso assoluto dell'oggetto File, e il percorso canonico. Quest'ultimo è un percorso completo in cui eventuali riferimenti relativi e simbolici sono già stati valutati e risolti. Quest'ultimo concetto ovviamente dipende fortemente dal sistema operativo.

```
public String getParent()
```

Restituisce il nome della *parent directory* dell'oggetto File. Per un file si tratta del nome della directory.

```
public boolean exists()
```

```
public boolean canWrite()
```

```
public boolean canRead()
```

Questi metodi permettono di capire se un file con il nome specificato esiste, se è scrivibile e se è leggibile.

```
public boolean isFile()
```

```
public boolean isDirectory()
```

```
public boolean isAbsolute()
```

Permettono di capire se l'oggetto File rappresenta un file, una directory o un percorso assoluto.

```
public long lastModified()
```

```
public long length()
```

Permettono di conoscere la data dell'ultima modifica del file, e la sua lunghezza in byte.

```
public boolean renameTo(File dest)

public boolean delete()
```

Permettono di rinominare e di cancellare un file.

```
public boolean mkdir()
public boolean mkdirs()
```

Permette di creare una directory che corrisponde all'oggetto File. La seconda versione crea se necessario tutte le directory del percorso dell'oggetto File.

```
public String[] list()
public String[] list(FilenameFilter filter)
```

Restituiscono l'elenco di tutti i file della directory corrispondente all'oggetto File. Nella seconda versione è possibile specificare un filtro.

## Descrizione classe RandomAccess

```
public class RandomAccessFile extends Object implements DataOutput, DataInput
```

Anche in Java è possibile accedere ai file in modo random, cioè non in modo sequenziale. Tramite questa classe infatti è possibile accedere a una particolare posizione in un file, ed è inoltre possibile accedere al file contemporaneamente in lettura e scrittura (cosa che non è possibile con l'accesso sequenziale messo a disposizione dagli stream sui file, che saranno illustrati successivamente). È comunque possibile specificare in che modo accedere a un file (solo lettura, o lettura e scrittura).

La classe RandomAccessFile, implementando le interfacce DataInput e DataOutput, rende possibile scrivere in un file tutti gli oggetti e i tipi di dati primitivi. La classe inoltre fornisce i metodi per gestire la posizione corrente all'interno del file.

Se si scrive su un file esistente ad una particolare posizione si sovrascriveranno i dati a quella posizione.

## Costruttori

```
public RandomAccessFile(String file, String mode) throws IOException

public RandomAccessFile(File file, String mode) throws IOException
```

Si può specificare il file da aprire sia tramite una stringa, sia tramite un oggetto File. Si deve inoltre specificare il modo di apertura del file nella stringa mode. Con la stringa "r" si apre il file in sola lettura, e con "rw" sia in lettura che in scrittura.

# Metodi

```
public int read() throws IOException
```

Legge un byte. Blocca il processo chiamante se non è disponibile dell'input.

```
public int read(byte b[], int off, int len) throws IOException
```

```
public int read(byte b[]) throws IOException
```

Riempie un array o una parte dell'array specificato con i dati letti dal file. Viene restituito il numero di byte effettivamente letti.

```
public final void readFully(byte b[]) throws IOException
```

```
public final void readFully(byte b[], int off, int len) throws IOException
```

Questi metodi cercano di riempire un array (o una sua parte) con i dati letti dal file. Se viene raggiunta la fine del file prima di aver terminato, viene lanciata un'eccezione EOFException.

```
public final FileDescriptor getFD() throws IOException
```

Viene restituito un descrittore di file utilizzato dal sistema operativo per gestire il file. Si tratta di un descrittore a basso livello rappresentato dalla classe FileDescriptor. Difficilmente ci sarà la necessità di gestire direttamente tale informazione.

```
public int skipBytes(int n) throws IOException
```

Questo metodo salta n byte, bloccandosi finché non sono stati saltati. Se prima di questo si incontra la fine del file, viene sollevata un'eccezione EOFException.

```
public void write(int b) throws IOException
public void write(byte b[]) throws IOException
public void write(byte b[], int off, int len) throws IOException
```

Questi metodi permettono di scrivere rispettivamente in un file un singolo byte (nonostante l'argomento sia di tipo intero, solo il byte meno significativo viene effettivamente scritto nel file), un intero array, o una parte.

```
public native long getFilePointer() throws IOException
```

Restituisce la posizione corrente all'interno del file, cioè la posizione in cui si sta leggendo o scrivendo.

```
public void seek(long pos) throws IOException
```

Sposta il puntatore all'interno del file alla posizione assoluta specificata in pos.

```
public long length() throws IOException
```

Restituisce la lunghezza del file.

```
public void close() throws IOException
```

Chiude il file (scrivendo sul disco eventuali dati bufferizzati).

Nella classe sono poi presenti diversi metodi per leggere e scrivere particolari tipi di dati (ad esempio readBoolean, writeBoolean, readInt, writeInt, ecc.), come quelli già visti nelle classi DataInputStream e DataOutputStream, del resto, come abbiamo visto, RandomAccessFile implementa le interfacce DataInput e DataOutput. Per una lista completa si faccia riferimento alla guida in linea.

## Le classi FileOutputStream e FileInputStream

Tramite queste classi è possibile accedere, rispettivamente in scrittura ed in lettura, sequenzialmente ai file, con il meccanismo degli stream.

## Descrizione classe FileOutputStream

```
public class FileOutputStream extends OutputStream
```

## Costruttori

```
public FileOutputStream(String name) throws IOException
public FileOutputStream(String name, boolean append) throws IOException
```

Si può aprire un file in scrittura specificandone il nome tramite una stringa. Se esiste già un file con lo stesso nome, verrà sovrascritto. È possibile (con il secondo costruttore) specificare se il file deve essere aperto in *append mode*.

```
public FileOutputStream(File file) throws IOException
```

Si può specificare il file da aprire tramite un oggetto File già esistente. Anche in questo caso, se il file esiste già, viene sovrascritto.

```
public FileOutputStream(FileDescriptor fdObj)
```

Si può infine specificare il file con cui collegare lo stream tramite un FileDescriptor. In questo modo si apre uno stream su un file già aperto, ad esempio uno aperto per accesso random. Ovviamente utilizzando questo costruttore non si crea (e quindi non si sovrascrive) un file, che anzi, come già detto, deve essere già aperto.

## Metodi

```
public final FileDescriptor getFD() throws IOException
```

In questo modo è possibile ottenere il FileDescriptor relativo al file collegato allo stream.

```
public native void close() throws IOException
```

Si dovrebbe chiamare sempre questo metodo quando non si deve più scrivere sul file. Questo metodo sarà comunque chiamato automaticamente quando lo stream sarà sottoposto al *garbage collecting*.

## Descrizione classe FileInputStream

```
public class FileInputStream extends InputStream
```

## Costruttori

```
public FileInputStream(String name) throws FileNotFoundException
public FileInputStream(File file) throws FileNotFoundException
public FileInputStream(FileDescriptor fdObj)
```

Uno stream può essere aperto specificando il file da aprire negli stessi modi visti nella classe FileOutputStream. Nel terzo caso il file è già aperto, ma nei primi due no: in tal caso il file deve esistere, altrimenti verrà lanciata un'eccezione FileNotFoundException.

## Metodi

```
public final FileDescriptor getFD() throws IOException
```

In questo modo è possibile ottenere il FileDescriptor relativo al file collegato allo stream.

Ecco ora un piccolo esempio di utilizzo di questi due stream per effettuare la copia di due file. Il nome dei due file (sorgente e destinazione) dovrà essere specificato sulla linea di comando.

```
import java.io.*;

public class CopyFile {
  static public void main (String args[]) throws IOException {
```

```
        if(args.length != 2){
          String Msg;
          Msg = "Sintassi: CopyFile <sorgente> <destinazione>"
          throw(new IOException(Msg));
        }

        FileInputStream in = new FileInputStream(args[0]);
        FileOutputStream out = new FileOutputStream(args[1]);

        byte buffer[] = new byte[256];
        int n;
        while((n = in.read (buffer)) > -1)
          out.write(buffer, 0, n);

        out.close();
        in.close();
    }
  }
```

# Classi Reader e Writer

Dalla versione 1.1 del JDK, sono stati introdotti gli stream che gestiscono i caratteri (*character stream*). Tutti gli stream esaminati fino ad adesso gestiscono solo byte; i character stream sono come i byte stream, ma gestiscono caratteri Unicode a 16 bit, invece che byte (8 bit). Le classi base della gerarchia di questi stream sono Reader e Writer; tali classi supportano le stesse operazioni che erano presenti in InputStream e OutputStream, tranne che per il fatto che, laddove i byte stream operano su byte e su array di byte, i character stream operano su caratteri, array di caratteri, o stringhe.

Il vantaggio degli stream di caratteri è che rendono i programmi indipendenti dalla particolare codifica dei caratteri del sistema su cui vengono eseguite le applicazioni (a tal proposito si veda anche il capitolo sull'internazionalizzazione).

Java infatti per memorizzare le stringhe utilizza l'Unicode; l'Unicode è una codifica con la quale è possibile rappresentare la maggior parte dei caratteri delle varie lingue. I character stream quindi rendono trasparente la complessità di utilizzare le varie codifiche, mettendo a disposizione delle classi che automaticamente provvedono a eseguire la conversione fra gli stream di byte e gli stream di caratteri. La classe InputStreamReader, ad esempio, implementa un input stream di caratteri che legge i byte da un input stream di byte e li converte in caratteri. Allo stesso modo un OutputStreamWriter implementa un output stream di caratteri che converte i caratteri in byte e li scrive in un output stream di byte. Per creare un InputStreamReader basterà quindi eseguire la seguente operazione:

```
InputStreamReader in = new InputStreamReader(System.in);
```

Inoltre gli stream di caratteri sono più efficienti dei corrispettivi stream di byte, in quanto,

mentre questi ultimi eseguono spesso operazioni di lettura e scrittura un byte alla volta, i primi tendono a utilizzare di più la bufferizzazione.

A tal proposito esistono anche le classi BufferedReader e BufferedWriter, che corrispondo a BufferedInputStream e BufferedOutputStream; si può quindi scrivere

```
BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
```

## Le classi PrintStream e PrintWriter

Della classe PrintStream si è già parlato all'inizio del capitolo quando si sono introdotti gli stream predefiniti in, out ed err. Tale classe, nel JDK 1.1, è stata modificata in modo da utilizzare la codifica dei caratteri della piattaforma sottostante. Quindi in realtà ogni PrintStream incorpora un OutputStreamWriter e utilizza tale stream per gestire in modo adeguato i caratteri da stampare.

Invece di rendere *deprecated* l'intera classe PrintStream, sono stati resi deprecati i suoi costruttori. In questo modo tutti i programmi esistenti che, per stampare informazioni di debug o errori sullo schermo, utilizzano il metodo System.out.println o System.err.println potranno essere compilati senza ottenere warning. Si otterrà invece un warning se si costruisce esplicitamente un PrintStream.

In questi casi si dovrebbe invece costruire un PrintWriter, a cui si può passare un OutputStream, e che provvederà automaticamente a utilizzare un OutputStreamWriter intermedio per codificare in modo corretto i caratteri da stampare. I metodi per stampare sono quelli di PrintStream e cioè print e println, in grado di gestire i vari tipi primitivi di Java.

# Altre classi e metodi deprecati

Quando si è trattato DataInputStream si è volutamente evitato il metodo readLine, per leggere una linea di testo dallo stream di input collegato, perché tale metodo è deprecated; questo è dovuto al fatto che non avviene la giusta conversione da byte a carattere. In tal caso si dovrebbe usare invece un BufferedReader, e il relativo metodo readLine.

Quindi, dato uno stream di input in, invece di creare un DataInputStream, se si vuole utilizzare il metodo readLine, si dovrà creare un BufferedReader:

```
BufferedReader d= new BufferedReader(new InputStreamReader(in));
```

Comunque si potrà continuare a utilizzare la classe DataInputStream per tutte le altre operazioni di lettura.

Anche la classe LineNumberInputStream, utilizzata per tenere traccia delle linee all'interno di uno stream tramite il metodo getLineNumber(), è deprecata; al suo posto si dovrà utilizzare un LineNumberReader.

# Programmazione concorrente e gestione del multithread in Java

Paolo Aiello, Giovanni Puliti

## Introduzione

Una delle potenti caratteristiche del linguaggio Java è il supporto per la *programmazione concorrente* o *parallela*. Tale *feature* permette di organizzare il codice di una stessa applicazione in modo che possano essere mandate in esecuzione contemporanea più parti di codice differenti fra loro.

Prima di descrivere questi aspetti del linguaggio saranno introdotti alcuni concetti fondamentali che aiuteranno ad avere un'idea più chiara dell'argomento e delle problematiche correlate.

## Processi e multitasking

Tutti i moderni sistemi operativi offrono il supporto per il *multitasking*, ossia permettono l'esecuzione simultanea di più *processi*. In un sistema Windows, Unix o Linux si può, ad esempio, scrivere una e-mail o un documento di testo mentre si effettua il download di un file da Internet. In apparenza questi diversi programmi vengono eseguiti contemporaneamente, anche se il computer è dotato di un solo processore.

In realtà i processori dei calcolatori su cui si è abituati a lavorare analizzano il flusso delle istruzioni in maniera sequenziale in modo che in ogni istante una sola istruzione sia presa in esame ed eseguita (questo almeno in linea di massima, dato che esistono architetture particolari che permettono il parallelismo a livello di microistruzioni).

Ma anche se, per sua natura, un computer è una macchina sequenziale, grazie a una gestione

ciclica delle risorse condivise (prima fra tutte il processore centrale), si ottiene una specie di parallelismo, che permette di simulare l'esecuzione contemporanea di più programmi nello stesso momento.

Grazie alla elevata ottimizzazione degli algoritmi di gestione di questo pseudoparallelismo, e grazie alla possibilità di un processo di effettuare certi compiti quando gli altri sono in pausa o non sprecano tempo di processore, si ha in effetti una simulazione del parallelismo fra processi, anche se le risorse condivise sono in numero limitato.

Nel caso in cui si abbiano diversi processori operanti in parallelo, è possibile che il parallelismo sia reale, nel senso che un processore potrebbe eseguire un processo mentre un altro processore esegue un diverso processo, senza ripartizione del tempo: in generale non è possibile però fare una simile assunzione, dato che normalmente il numero di processi in esecuzione è maggiore (o comunque può esserlo) del numero di processori fisici disponibili, per cui è sempre necessario implementare un qualche meccanismo di condivisione delle risorse.

Un processo è un flusso di esecuzione del processore corrispondente a un programma. Il concetto di processo va però distinto da quello di programma in esecuzione, perché è possibile che un processore esegua contemporaneamente diverse istanze dello stesso programma, ossia generi diversi processi che eseguono lo stesso programma (ad esempio diverse istanze del Notepad, con documenti diversi, in ambiente Windows).
Per multitasking si intende la caratteristica di un sistema operativo di permettere l'esecuzione contemporanea (o pseudocontemporanea, per mezzo del time-slicing) di diversi processi.

Vi sono due tipi di multitasking:

il *cooperative multitasking* la cui gestione è affidata agli stessi processi, che mantengono il controllo del processore fino a che non lo rilasciano esplicitamente. Si tratta di una tecnica abbastanza rudimentale in cui il funzionamento dipende dalla bontà dal codice del programma, quindi in sostanza dal programmatore. C'è sempre la possibilità che un programma scritto in modo inadeguato monopolizzi le risorse impedendo il reale funzionamento multitasking. Esempi di sistemi che usano questo tipo di multitasking sono Microsoft Windows 3.x e alcune versioni del MacOS;

il *preemptive multitasking* è invece gestito interamente dal sistema operativo con il sistema del *time-slicing* (detto anche *time-sharing*), assegnando ad ogni processo un intervallo di tempo predefinito, ed effettuando il cambio di contesto anche senza che il processo intervenga o ne sia a conoscenza. Il processo ha sempre la possibilità di rilasciare volontariamente le risorse, ma questo non è necessario per il funzionamento del sistema. Il sistema operativo in questo caso utilizza una serie di meccanismi per il controllo e la gestione del tempo del processore, in modo da tener conto di una serie di parametri, legati al tempo trascorso e all'importanza (*priorità*) di un determinato processo.

Nonostante il fatto che i termini preemptive e time-slicing abbiano significato simile, in realtà preemptive si riferisce alla capacità di un processo di "prevalere" su un altro di minore priorità sottraendogli il processore in base a tale "diritto di priorità", mentre il time-slicing, anche se generalmente coesiste con la preemption, si riferisce unicamente alla suddivisione del tempo gestita dal sistema (e non lasciata ai processi), anche tra processi a priorità uguale. Lo scheduling usato per gestire i processi a uguale priorità è generalmente il cosiddetto round-robin scheduling, in cui un processo, dopo che ha usufruito della sua porzione di tempo, viene messo in attesa in coda fra processi con uguale priorità. Sia la preemption che il time-slicing presuppongono un intervento da parte del sistema operativo nel determinare quale processo deve essere mandato in esecuzione. Comunque possono esserci sistemi preemptive che non usano il time-slicing, ma usano ugualmente le priorità per determinare quale processo deve essere eseguito. Si tornerà su questo aspetto a proposito della gestione dei thread in Java.

Si è detto che per simulare il parallelismo fra processi differenti si effettua una spartizione del tempo trascorso in esecuzione nel processore. Il meccanismo di simulazione si basa sul cambio di contesto (*context-switch*) fra processi diversi: in ogni istante un solo processo viene messo in esecuzione, mentre gli altri restano in attesa.

Il contesto di un processo P1 è l'insieme delle informazioni necessarie per ristabilire esattamente lo stato in cui si trova il sistema al momento in cui interrompe l'esecuzione del processo P1 per passare a un altro processo P2. Tra queste informazioni di contesto le principali sono lo *stato dei registri del processore*, e la *memoria del processo*, che a sua volta contiene il *testo* del programma, ossia la sequenza di istruzioni, i *dati* gestiti dal processo e lo *stack* (spazio di memoria per le chiamate di funzioni e le variabili locali).

Infatti, un aspetto fondamentale della gestione dei processi è il fatto che *ogni processo ha un suo spazio di memoria privato*, a cui esso soltanto può accedere. Quindi, salvo casi eccezionali (memoria condivisa) un processo non ha accesso alla memoria gestita da un altro processo.

I processi sono normalmente organizzati secondo una struttura gerarchica in cui, a partire da un primo processo iniziale creato alla partenza del sistema operativo, ogni successivo processo è "figlio" di un altro processo che lo crea e che ne diviene il "padre".

Nei sistemi preemptive vi è poi un processo particolare che gestisce tutti gli altri processi, lo *scheduler*, responsabile della corretta distribuzione del tempo della CPU tra i processi in esecuzione. A tale scopo esistono diversi algoritmi di *scheduling*, che comunque generalmente si basano sul tempo di attesa (maggiore è il tempo trascorso dall'ultima esecuzione, maggiore è la priorità del processo) e su livelli di priorità intrinseci assegnati dal sistema sulla base della natura del processo, oppure dall'utente sulla base delle sue esigenze particolari. A prescindere da questo normale avvicendamento di esecuzione, i processi possono subire delle interruzioni (*interrupt*) dovute al verificarsi di eventi particolari, originati dall'hardware come l'input di una periferica (interrupt hardware), dal software (interrupt software) oppure da errori di esecuzione che causano le cosiddette *eccezioni*. In questi casi viene effettuato un context-switch come nel normale scheduling, viene eseguito del codice specifico che gestisce l'interruzione, dopodiché si torna al processo interrotto con un altro context-switch. I processi, durante il loro ciclo di vita, assumono *stati* differenti, in conseguenza del loro funzionamento interno

e dell'attività dello scheduler. Semplificando al massimo, questi sono i principali stati che un processo può assumere:

- *in esecuzione*: il processo è attualmente in esecuzione;
- *eseguibile*: il processo non è in esecuzione, ma è pronto per essere eseguito, appena la CPU si rende disponibile;
- *in attesa*: il processo è in attesa di un dato evento, come lo scadere di una frazione di tempo, la terminazione di un altro processo, l'invio di dati da un canale I/O.

I processi normalmente sono entità tra loro separate ed estranee ma, qualora risultasse opportuno, sono in grado di comunicare tra di loro utilizzando mezzi di comunicazione appositamente concepiti, genericamente identificati dalla sigla IPC (Inter Process Communication). Tra questi si possono citare la memoria condivisa (*shared-memory*), i *pipe*, i *segnali*, i *messaggi*, i *socket*. A seconda della tipologia di comunicazione tra processi, possono sorgere dei problemi derivanti dall'acceso contemporaneo, diretto o indiretto, alle medesime risorse. Per evitare che questo dia origine a errori e incongruenze, generalmente le risorse vengono acquisite da un singolo processo con un *lock*, e rilasciate una volta che l'operazione è terminata. Solo a quel punto la risorsa sarà disponibile per gli altri processi. Per gestire questo tipo di problemi di *sincronizzazione* esistono appositi meccanismi, tra cui il più conosciuto è quello dei *semafori*. Per maggiori approfondimenti legati a questi argomenti si faccia riferimento alla bibliografia.

## Thread e multithreading

L'esecuzione parallela e contemporanea di più tasks (intendendo per task l'esecuzione di un compito in particolare), risulta utile non solo nel caso di processi in esecuzione su un sistema operativo multitasking, ma anche all'interno di un singolo processo.

Si pensi, ad esempio, a un editor di testo in cui il documento corrente viene automaticamente salvato su disco ogni *n* minuti. In questo caso il programma è composto da due flussi di esecuzione indipendenti tra loro: da un lato l'editor che raccoglie i dati in input e li inserisce nel documento, dall'altra il meccanismo di salvataggio automatico che resta in attesa per la maggior parte del tempo e, a intervalli prestabiliti, esegue la sua azione.

Sulla base di simili considerazioni è nata l'esigenza di poter usare la programmazione concorrente all'interno di un singolo processo, e sono stati concepiti i *thread*, i quali in gran parte replicano il modello dei processi concorrenti, applicato però nell'ambito di una singola applicazione. Un processo quindi non è più un singolo flusso di esecuzione, ma un insieme di flussi: ogni processo contiene almeno un thread (thread principale) e può dare origine ad altri thread generati a partire dal thread principale.

Come per il multitasking, anche nel multithreading lo scheduling dei thread può essere compiuto dal processo (dal thread principale), eventualmente appoggiandosi ai servizi offerti dal sistema operativo, se questo adotta il *time-slicing*; in alternativa può essere affidato ai singoli thread, ed allora il programmatore deve fare attenzione a evitare che un singolo thread mono-

polizzi le risorse, rilasciandole periodicamente secondo criteri efficienti.

La differenza fondamentale tra processi e thread sta nel fatto che i thread *condividono lo stesso spazio di memoria*, se si prescinde dallo stack, ossia dai dati temporanei e locali usati dalle funzioni.

Questo porta diverse conseguenze: il cambio di contesto fra thread è molto meno pesante di quello tra processi, e quindi l'uso di thread diversi causa un dispendio di risorse inferiore rispetto a quello di processi diversi; inoltre la comunicazione fra thread è molto più semplice da gestire, dato che si ha condivisione dello stesso spazio di memoria.

D'altra parte, proprio questa condivisione rende molto più rilevanti e frequenti i problemi di sincronizzazione, come si vedrà dettagliatamente in seguito.

Un thread è un flusso di esecuzione del processore corrispondente a una sequenza di istruzioni all'interno di un processo. Analogamente ai processi, bisogna distinguere il concetto di esecuzione di una sequenza di istruzioni da quello di thread, poiché ci possono essere diverse esecuzioni parallele di uno stesso codice, che danno origine a thread diversi.

Per multithreading si intende l'esecuzione contemporanea (ovvero pseudocontemporanea, per mezzo del time-sharing) di diversi thread nell'ambito dello stesso processo. La gestione del multithreading può essere a carico del sistema operativo, se questo supporta i thread, ma può anche essere assunta dal processo stesso.

# I thread e la Java Virtual Machine

Si è visto che un thread è un flusso di esecuzione nell'ambito di un processo. Nel caso di Java, ogni esecuzione della macchina virtuale dà origine a un processo, e tutto quello che viene mandato in esecuzione da una macchina virtuale (ad esempio un'applicazione o una Applet) dà origine a un thread.

La virtual machine Java è però un processo un po' particolare, in quanto funge da ambiente portabile per l'esecuzione di applicazioni su piattaforme differenti. Quindi la JVM non può fare affidamento su un supporto dei thread da parte del sistema operativo, ma deve comunque garantire un certo livello minimo di supporto, stabilito dalle specifiche ufficiali della virtual machine. Queste stabiliscono che una VM gestisca i thread secondo uno scheduling di tipo preemptive chiamato *fixed-priority scheduling*. Questo schema è basato essenzialmente sulla priorità ed è preemptive perché garantisce che, se in qualunque momento si rende eseguibile un thread con priorità maggiore di quella del thread attualmente in esecuzione, il thread a maggiore priorità prevalga sull'altro, assumendo il controllo del processore.

La garanzia che sia sempre in esecuzione il thread a priorità più alta non è assoluta. In casi particolari lo scheduler può mandare in esecuzione un thread a priorità più bassa per evitare situazioni di stallo o un consumo eccessivo di risorse. Per questo motivo è bene non fare affidamento su questo comportamento per assicurare il corretto funzionamento di un algoritmo dal fatto che un thread ad alta priorità prevalga sempre su uno a bassa priorità.

Le specifiche della VM non richiedono il time-slicing nella gestione dei thread, anche se questo è in realtà presente nei più diffusi sistemi operativi e, di conseguenza può essere utilizzato dalle VM che girano su questi sistemi. Per questo motivo, se si vuole che un'applicazione Java funzioni correttamente indipendentemente dal sistema operativo e dalla implementazione della VM, non si deve assumere la gestione del time-sharing da parte della VM, ma bisogna far sì che ogni thread rilasci spontaneamente le risorse quando opportuno. Quest'aspetto sarà analizzato nei dettagli più avanti.

Si diceva che generalmente le VM usano i servizi di sistema relativi ai thread, se presenti. Ma ciò non è tassativo. Una macchina virtuale può anche farsi interamente carico della gestione dei thread, senza far intervenire il sistema operativo, anche se questo supporta i thread.In questo caso la VM è vista dal sistema come un processo con un singolo thread, mentre i thread Java sono ignorati totalmente dal sistema stesso. Questo modello di implementazione dei thread nella VM è conosciuto come *green-thread* (*green* in questo caso è traducibile approssimativamente con *semplice*) ed è adottato da diverse implementazioni della VM, anche in sistemi (ad esempio alcune versioni di Unix) in cui esiste un supporto nativo dei thread. Viceversa in ambiente Windows, le VM usano generalmente i servizi del sistema operativo. Analogamente ai processi, i thread assumono in ogni istante un determinato stato. Nella VM si distinguono i seguenti stati dei thread:

- *initial*: un thread si trova in questa condizione tra il momento in cui viene creato e il momento in cui comincia effettivamente a funzionare;

- *runnable*: è lo stato in cui si trova normalmente un thread dopo che ha cominciato a funzionare. Il thread in questo stato può, in qualunque momento, essere eseguito;

- *running*: il thread è attualmente in esecuzione. Questo non sempre viene considerato uno stato a sé, ma in effetti si tratta di una condizione diversa dallo stato runnable. Infatti ci possono essere diversi thread nello stato runnable in un dato istante ma, in un sistema a singola CPU, uno solo è in esecuzione, e viene chiamato *thread corrente*;

- *blocked*: il thread è in attesa di un determinato evento;

- *dead*: il thread ha terminato la sua esecuzione.

# La programmazione concorrente in Java

Dopo tale panoramica su programmazione parallela e thread, si può analizzare comeutilizzare i thread in Java. Gli strumenti a disposizione per la gestione dei thread sono essenzialmente due: la classe `java.lang.Thread` e l'interfaccia `java.lang.Runnable`.

Dal punto di vista del programmatore, i thread in Java sono rappresentati da oggetti che sono o istanze della classe `Thread`, o istanze di una sua sottoclasse, oppure oggetti che implementano l'interfaccia `Runnable`. D'ora in avanti si utilizzerà il termine thread sia per indicare il concetto di thread, sia per far riferimento alla classe `Thread` che a una qualsiasi classe che implementi le funzionalità di un thread.

## Creazione e terminazione di un thread

Inizialmente verrà presa in esame la modalità di creazione e gestione dei thread basata sull'utilizzo della classe Thread, mentre in seguito sarà analizzata la soluzione alternativa basata sull'interfaccia Runnable.

La classe Thread è una classe *non astratta* attraverso la quale si accede a tutte le principali funzionalità per la gestione dei thread, compresa la creazione dei thread stessi. Il codice necessario per creare un thread è il seguente:

```
Thread myThread = new Thread();
```

A meno di associarvi un oggetto Runnable, istanziando direttamente un oggetto della classe Thread però non si ottiene nessun particolare risultato, dato che esso termina il suo funzionamento quasi subito: infatti le operazioni svolte in modalità threaded sono quelle specificate nel metodo run(), metodo che deve essere ridefinito dalle classi derivate.

Se si desidera quindi che il thread faccia qualcosa di utile ed interessante, si deve creare una sottoclasse di Thread, e ridefinire il metodo run(). Qui di seguito è riportato un esempio

```
public class SimpleThread extends Thread {
 String message;

 public SimpleThread(String s){
 message = s;
 }

 public void run() {
   for (int i = 0; i < 100; i++)
   System.out.println(message);
 }

 public static void main(String[] args) {
   SimpleThread st1, st2;
   st1 = new SimpleThread("Buongiorno");
   st2 = new SimpleThread("Buonasera");
   st1.start();
   st2.start();
 }
}
```

Per far partire il thread, una volta creato, si usa il metodo start(), il quale provoca l'esecuzione del metodo run(); terminato tale metodo, il thread cessa la sua attività, e le risorse impegnate per quel thread vengono rilasciate. A questo punto, se all'oggetto Thread è collegata a una variabile in uno scope ancora attivo, l'oggetto non viene eliminato dal garbage collector a meno che la variabile non venga impostata a null.

Tuttavia tale oggetto non è più utilizzabile, ed una successiva chiamata del metodo start(), pur non generando alcuna eccezione, non avrà alcun effetto; la regola di base dice infatti che l'oggetto Thread è concepito per essere usato una volta soltanto.

È quindi importante tener presente che, se si hanno uno o più riferimenti, si dovrebbe aver cura di impostare tali variabili a null per liberare la memoria impegnata dall'oggetto. Se invece creiamo il Thread senza alcun riferimento a una variabile, ad esempio

```
new SimpleThread("My Thread").start();
```

la virtual machine si fa carico di mantenere l'oggetto in memoria per tutta la durata di esecuzione del thread, e di renderlo disponibile per la garbage collection una volta terminata l'esecuzione.

La classe Thread contiene anche un metodo stop(), che permette di terminare l'esecuzione del thread dall'esterno. Ma questo metodo è deprecato in Java 2 per motivi di sicurezza. Infatti in questo caso l'esecuzione si interrompe senza dare la possibilità al thread di eseguire un cleanup: il thread in questo caso non ha alcun controllo sulle modalità di terminazione. Per questo motivo l'uso di stop() è da evitare comunque, indipendentemente dalla versione di Java che si usa.

## L'interfaccia Runnable

L'altra possibilità che permette di creare ed eseguire thread si basa sull'utilizzo della interfaccia Runnable a cui si accennava in precedenza. Ecco un esempio, equivalente al precedente, ma che usa una classe Runnable anziché una sottoclasse di Thread:

```
public class SimpleRunnable implements Runnable {
 String message;

 public SimpleRunnable(String s) {
   message = s;
 }

 public void run() {
   for (int i = 0; i < 100; i++)
     System.out.println(message);
 }

 public static void main(String[] args) {
   SimpleRunnable sr1, sr2;
   sr1 = new SimpleRunnable(„Buongiorno");
   sr2 = new SimpleRunnable("Buonasera");
```

```
        Thread t1 = new Thread(sr1);
        Thread t2 = new Thread(sr2);
        t1.start();
        t2.start();
    }
}
```

L'interfaccia Runnable contiene un solo metodo, il metodo run(), identico a quello già visto per la classe Thread. Questa non è una semplice coincidenza, dal momento che la classe Thread, in realtà, implementa l'interfaccia Runnable.

Per la precisione implementando l'interfaccia Runnable e il metodo run(), una classe *non derivata da* Thread può funzionare come un Thread, e per far questo però, deve essere "agganciata" a un oggetto Thread (un'istanza della classe Thread o di una sua sottoclasse) passando un reference dell'oggetto Runnable al costruttore del Thread.

Dall'esempio fatto, però, l'interfaccia Runnable non risulta particolarmente utile, anzi sembra complicare inutilmente le cose: che bisogno c'è di rendere un altro oggetto Runnable se si può usare direttamente una sottoclasse di Thread?

Per rispondere a tale quesito, si pensi ad una situazione in cui si voglia far sì che una certa classe implementi contemporaneamente la funzione di thread, ma che specializzi anche un'altra classe base (fig. 10.1).

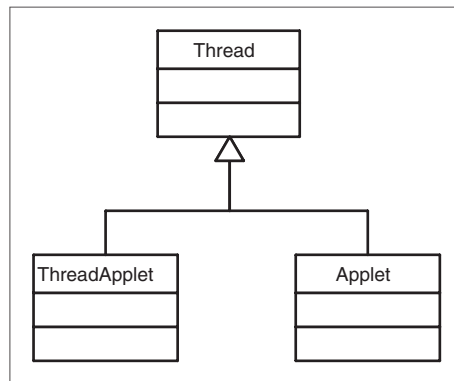
**Figura 10.1** – *Per creare una classe che sia contemporaneamente un Thread ma anche qualcos'altro, si può optare per una ereditarietà multipla. Tale soluzione non è permessa in Java.*

Ora dato che in Java non è permessa l'ereditarietà multipla, tipicamente una soluzione a cui si ricorre è quella di utilizzare uno schema progettuale differente, basato spesso sul pattern Delegation o sullo Strategy (come mostrato in fig. 10.2).

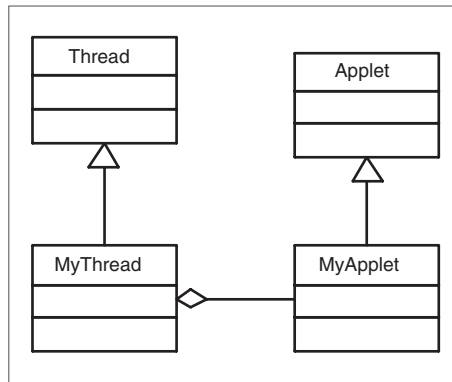
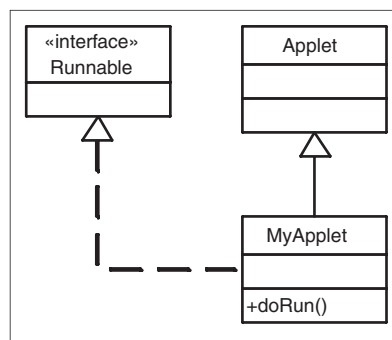
**Figura 10.2** – *In alternativa, si può ereditare da un solo padre ed inglobare un oggetto che svolga la funzione che manca. Questo pattern, molto utilizzato, non risulta essere particolarmente indicato nel caso dei thread.*

Questa architettura non si adatta molto bene al caso dei thread, o comunque risulta essere troppo complessa, visto che l'interfaccia `Runnable` ne offre un'altra molto più semplice. Derivando dalla classe base e implementando l'interfaccia `Runnable` infatti si può sia personalizzare la classe base, sia aggiungere la funzione di thread (si veda la fig. 10.3).

**Figura 10.3** – *Grazie all'utilizzo dell'interfaccia Runnable, si possono derivare classi al fine di specializzarne il comportamento ed aggiungere funzionalità di thread.*

Ecco con un esempio come si può implementare tale soluzione

```
class RunnableApplet extends Applet implements Runnable {
 String message;
```

```
 RunnableApplet(String s) {
   message = s;
 }

 public void init() {
   Thread t = new Thread(this);
   t.start();
 }

 public void run() {
   for (int i = 0; i < 100; i++)
     System.out.println(message);
}
```

Anche se l'esempio è forse poco significativo, riesce a far capire come l'oggetto può eseguire nel metodo run() dei compiti suoi propri, usando i suoi dati e i suoi metodi, e anche quelli ereditati dalla classe base, mentre l'oggetto Thread incapsulato viene usato solo per eseguire tutto questo in un thread separato.

Utilizzando una variabile di classe per il thread possiamo incrementare il controllo sul thread: aggiungendo un metodo start() è possibile far partire il thread dall'esterno, al momento voluto anziché automaticamente in fase di inizializzazione dell'Applet:

```
class RunnableApplet extends Applet implements Runnable {
 String message;
 Thread thread;

 RunnableApplet(String s) {
   message = s;
 }

 public void init() {
   thread = new Thread(this);
 }

 public void start() {
   t.start();
 }

 public void run() {
   for (int i = 0; i < 100; i++)



System.out.println(message);
}
```

Questo è il caso più tipico di utilizzo dell'interfaccia Runnable: un oggetto Thread viene inglobato in un oggetto già derivato da un'altra classe e utilizzato come "motore" per l'esecuzione di un certo codice in un thread separato.

Quindi si può dire semplicisticamente che l'uso dell'interfaccia Runnable al posto della derivazione da Thread si rende necessario quando la classe che si vuole rendere Runnable è già una classe derivata.

Negli altri casi si può scegliere il metodo che appare più conveniente.

## Identificazione del thread

Ogni thread che viene creato assume un'identità autonoma all'interno del sistema: per facilitarne la successiva identificazione è possibile assegnare un nome al thread, passandolo al costruttore. Ad esempio:

```
SimpleRunnable sr = new SimpleRunnable("Buongiorno");
Thread t = new Thread (sr, "Thread che saluta");
```

con una successiva chiamata del metodo getName() è possibile conoscere il nome del thread.

In ogni caso se non è stato assegnato al momento della creazione, il runtime Java provvede ad assegnare a ciascun thread un nome simbolico che però non è molto esplicativo all'occhio dell'utente.

L'uso di nomi significativi è particolarmente utile in fase di debugging, rendendo molto più facile individuare e selezionare il thread che si vuol porre sotto osservazione.

# Maggior controllo sui thread

Oltre alla gestione ordinaria dei thread, Java fornisce una serie di strumenti che permettono di gestire l'esecuzione di un thread fin nei minimi dettagli. Se da un lato questo permette una maggiore capacità di controllo del thread stesso, dall'altro comporta un miglior controllo sulle risorse che sono utilizzate durante l'esecuzione.

Di conseguenza migliora il livello di portabilità della applicazione, dato che si può sopperire a certe carenze del sistema operativo.

## Una fine tranquilla: uscire da run()

Negli esempi precedenti sono stati presi in considerazione casi con thread che eseguono un certo compito per un lasso limitato di tempo (stampare un certo messaggio 100 volte). Finito il compito, il thread termina e scompare dalla circolazione.

Spesso accade però che un thread possa vivere per tutta la durata dell'applicazione e svolgere il suo compito indefinitamente, senza mai terminare; oppure continui finché il suo funzionamento non venga fatto cessare volutamente.

Un esempio tipico potrebbe essere quello che segue: un "thread-orologio" mostra in questo caso l'ora corrente aggiornandola periodicamente:

```
public class Clock extends Applet implements Runnable {
 private Thread clockThread = null;

 public void start() {
    if (clockThread == null) {
       clockThread = new Thread(this, "Clock");
       clockThread.start();
    }
 }

 public void run() {
    while (clockThread != null) {
       repaint();
    try {
       // rimane in attesa per un secondo
       Thread.sleep(1000);
    }
    catch (InterruptedException e) {}
    }
 }

 public void paint(Graphics g) {
    // prende data e ora corrente
    Calendar systemTime = Calendar.getInstance();
    // formatta e visualizza l'ora
    DateFormat formatter = new SimpleDateFormat("HH:mm:ss");
    g.drawString(formatter.format(systemTime.getTime()), 5, 10);
 }

 public void stop() {
    clockThread = null;
 }
}
```

Al momento della creazione e inizializzazione della Applet (alla visualizzazione della pagina HTML nel browser) viene creato e fatto partire un thread.

Quando la pagina che contiene l'Applet viene lasciata, viene eseguito il metodo stop(), che mette la variabile clockThread a null. Ciò ha un doppio effetto: determina la terminazione del thread (quando la variabile ha il valore null il ciclo while del metodo run ha termine), e rende disponibile la memoria dell'oggetto Thread per la garbage collection.

## Bisogno di riposo: il metodo sleep()

Riprendendo in esame la classe Clock, si può notare che nel metodo run() viene chiamato il metodo repaint(), che a sua volta determina l'esecuzione del metodo paint(), il quale infine visualizza l'ora di sistema.

Se non si utilizzassero ulteriori accorgimenti si avrebbe un grande spreco di risorse e un funzionamento tutt'altro che ottimale: l'ora infatti sarebbe continuamente aggiornata, senza alcuna utilità dal momento che vengono visualizzati solo i secondi, causando per di più un rallentamento del sistema e uno sfarfallio dell'immagine (effetto flickering).

Per evitare tutto questo viene in aiuto il metodo `sleep()`, che permette di sospendere l'esecuzione di un thread (facendolo passare allo stato blocked) per un periodo di tempo prefissato, specificato in millisecondi. Nel caso in questione, una sospensione della durata di un secondo è esattamente l'intervallo sufficiente per l'aggiornamento dell'orologio.

Una chiamata del metodo `sleep()` provoca una messa in attesa del thread corrente e l'esecuzione del primo thread in attesa, con conseguente cambio di contesto, non previsto dalla tabella dello scheduler. Questa operazione da una parte comporta un certo costo computazionale (che va tenuto presente), ma dall'altra libera una risorsa che talvolta, come nel nostro esempio, rimarrebbe inutilmente occupata.

Si tenga presente che la sospensione del thread per mezzo di una `sleep` può essere pericolosa nel caso in cui si implementi una qualche gestione sincronizzata delle variabili (vedi oltre), dato che non rilascia gli eventuali lock acquisti dal thread.

Il metodo `sleep()` è un metodo statico della classe `Thread`, e ha come effetto di sospendere l'esecuzione del thread corrente. Di conseguenza è possibile chiamarlo da qualunque classe, anche se non viene usato alcun oggetto di tipo `Thread`.

## Gioco di squadra: il metodo yield()

Si è visto precedentemente, parlando dei processi, che esiste una forma di multitasking chiamato *cooperativo*, in cui ogni processo cede volontariamente il controllo del processore, dato che il sistema non gestisce lo scheduling dei processi. Si è anche detto che le specifiche della virtual machine non prevedono il time-slicing per cui, in presenza di thread di uguale priorità non è garantito che un thread che non rilasci le risorse di sua iniziativa non resti in esecuzione indefinitamente, impedendo di fatto agli altri thread di funzionare.

Per questi motivi, normalmente è buona norma non definire blocchi di istruzioni che possono richiedere molto tempo per essere eseguite ma, in alternativa, spezzare tali blocchi in entità più piccole. Lo scopo è quello di facilitare il compito dell'algoritmo di "schedulazione" in modo da evitare che un solo thread monopolizzi il processore per periodi troppo lunghi.

Anche nel caso in cui il sistema si faccia carico di partizionare il tempo di esecuzione, spesso lo scheduler non è in grado di stabilire in maniera automatica dove e quando risulti più opportuno interrompere un thread.

Il metodo `yield()` permette di gestire in maniera ottimale queste situazioni: esso consente infatti di cedere l'uso del processore a un altro thread in attesa con il grosso vantaggio che, nel caso in cui nessuno sia in attesa di essere servito, permette il proseguimento delle operazioni del thread invocante senza un inutile e costoso cambio di contesto.

L'invocazione di `yield()` non provoca un cambio di contesto (il thread rimane runnable), ma piuttosto viene spostato alla fine della coda dei thread della sua stessa priorità.

Ciò significa che questo metodo ha effetto solo nei confronti di altri thread di uguale priorità,

dato che i thread a priorità inferiore non prendono il posto del thread corrente anche se questo usa il metodo yield().

Utilizzando yield è il programmatore che stabilisce come e dove è opportuno cedere il processore, indipendentemente da quello che è poi il corso storico dei vari thread.

È bene eseguire una chiamata a tale funzione in quei casi in cui si ritiene che il thread possa impegnare troppo a lungo il processore, in modo da facilitare la cooperazione fra thread, permettendo una migliore gestione delle risorse condivise.

Il metodo yield() è un metodo statico della classe Thread, e ha effetto sul thread corrente. È possibile quindi chiamarlo da qualunque classe senza riferimento a un oggetto di tipo Thread.

## La legge non è uguale per tutti: la priorità

Si è visto che la virtual machine adotta uno scheduling di tipo preemptive, basato sulla priorità: ogni volta quindi che un thread di priorità maggiore del thread in esecuzione diventa runnable, si ha un cambio di contesto; per questo in linea di massima il thread corrente è sempre un thread a priorità più alta.

Si è anche detto che la virtual machine non prevede necessariamente il time-slicing ma, se questo è presente, i thread a maggiore priorità dovrebbero occupare la CPU per un tempo maggiore rispetto a quelli a minore priorità. L'esempio che segue mostra questi aspetti dei thread, illustrando l'uso dei metodi setPriority() e getPriority(); la classe CounterThread rappresenta il thread di base, utilizzato in seguito dalla ThreadPriority.

```
private class CounterThread extends Thread {
 boolean terminated = false;
 int count = 0;

 public void run() {
   while (!terminated) {
     count++;
     for (int i = 0; i < 1000; i++) {
       ;


   }
 }

 public void terminate() {
   terminated = true;
 }

 public int getCount() {
   return count;
 }
}
```

La classe che implementa Runnable, oltre ad utilizzare il thread precedente, imposta anche le priorità.

```
public class ThreadPriority implements Runnable {
 CounterThread thread1 = new CounterThread();
 CounterThread thread2 = new CounterThread();
 Thread thisThread = new Thread(this);
 int duration;

 public ThreadPriority(int priority1, int priority2,
 int duration) {
    this.duration = duration;
    thisThread.setPriority(Thread.MAX_PRIORITY);
    thread1.setPriority(priority1);
    thread2.setPriority(priority2);
    thread1.start();
    thread2.start();
    thisThread.start();
 }


 public void run() {
    try {
       for (int i = 0; i < duration; i++){
          System.out.println("Thread1: priority: "
+ thread1.getPriority()
 + " count: " + thread1.count);
          System.out.println("Thread2: priority: "
+ thread2.getPriority()
 + " count: " + thread2.count);
          thisThread.sleep(1000);
       }
    }
    catch (InterruptedException e){}

    thread1.terminate();
    thread2.terminate();

 }

 public static void main(String[] args) {
    new ThreadPriority(Integer.parseInt(args[0]),
  Integer.parseInt(args[1]),
  Integer.parseInt(args[2]));
 }
}
```

La classe ThreadPriority crea due oggetti CounterThread e li manda in esecuzione; successivamente manda in esecuzione il suo thread collegato (si tratta di una classe che implementa Runnable), il quale stampa i valori delle priorità e del counter dei thread ogni secondo (scheduler permettendo), e alla fine termina i due thread.

I valori di priorità e la durata in secondi sono dati come argomenti del main sulla linea di comando. I valori di priorità devono essere numeri interi da 1 a 10.

Si può notare che si è assegnata una priorità massima a thisThread, che deve poter interrompere gli altri due thread per eseguire la stampa e le chiamate terminate(): per fare questo si è usata la costante Thread.MAX_PRIORITY, che ha un valore uguale a 10.

La classe CounterThread aggiorna un contatore dopo aver eseguito un ciclo vuoto di 1000 iterazioni (ovviamente consumando una quantità abnorme di tempo della CPU, ma ai fini dell'esempio sorvoliamo su quest'aspetto).

Mandando in esecuzione il programma si può notare che effettivamente dopo un certo tempo il programma termina e vengono stampate le informazioni, il che significa che i due CounterThread sono stati interrotti dall'altro thread a priorità massima.

Se il sistema operativo e la VM supportano il time-slicing, il numero raggiunto dal contatore è approssimativamente proporzionale alla priorità del thread. Bisogna tener presente che possono esserci variazioni anche notevoli dato che i thread possono essere gestiti secondo algoritmi abbastanza complessi e variabili da implementazione a implementazione.

Tuttavia si nota che comunque, aumentando la priorità, aumenta il valore del contatore, e viceversa.

L'uso della gestione diretta delle priorità risulta molto utile in particolare nei casi in cui si ha un thread che resta nello stato blocked per la maggior parte del tempo. Assegnando a questo thread una priorità elevata si evita che rimanga escluso dall'uso della CPU in sistemi che non utilizzano il time-slicing. Questo è particolarmente importante per operazioni temporizzate che devono avere una certa precisione.

In casi in cui un certo thread compie delle operazione che fanno un uso intenso della CPU, e di lunga durata, abbassando la priorità del thread si disturba il meno possibile l'esecuzione degli altri thread. Ciò, ovviamente, sempre a patto che sia possibile mettere in secondo piano tale task.

## E l'ultimo chiuda la porta: il metodo join()

Il metodo join resta semplicemente in attesa finché il thread per il quale è stato chiamato non termina la sua esecuzione. Con questo metodo è quindi possibile eseguire una determinata operazione nel momento in cui un thread termina la sua esecuzione. Risulta pertanto utile in tutti i casi in cui un thread compie delle operazioni che utilizzano dei risultati dell'elaborazione di un altro thread.

Di seguito è riportato un breve esempio nel quale è mostrato come utilizzare tale metodo. In esso si è utilizzata una versione modificata della classe CounterThread, in cui è possibile specificare, come parametro del costruttore, il massimo valore raggiungibile dal counter. In tal modo possiamo limitare la durata di esecuzione del thread senza dover ricorrere al metodo terminate().

```
private class CounterThread extends Thread {
 boolean terminated = false;
 int count = 0;
 int maxCount = Integer.MAX_VALUE;

 public CounterThread() {
 }

 public CounterThread(int maxCount) {
   this.maxCount = maxCount;
 }

 public void run() {
   while (!terminated) {
     count++;
     for (int i = 0; i < 1000; i++) {
       // fai qualcosa
   }
 }

 public void terminate() {
   terminated = true;
 }

 public int getCount() {
   return count;
 }
}
```

La classe Chronometer misura il tempo di esecuzione, in minuti, secondi e millisecondi, di un thread che viene dato come argomento al metodo run(). Il metodo join() consente di determinare l'istante in cui termina l'esecuzione del thread (ovviamente con una certa approssimazione), e quindi di misurare il tempo trascorso dall'inizio dell'esecuzione.

```
public class Chronometer{
 Calendar startTime;
 Calendar endTime;

 public void run(Thread thread) {
   // registra l'ora di sistema all'inizio dell'esecuzione
   startTime = Calendar.getInstance();
   // manda in esecuzione il thread
   thread.start();
   try {
     // attende la fine dell'esecuzione
     thread.join();
   }
```

```
      catch (InterruptedException e) {}
      // registra l'ora di sistema alla fine dell'esecuzione
      endTime = Calendar.getInstance();
    }

    // calcola il tempo trascorso e restituisce
    // una stringa descrittiva
    public String getElapsedTime() {
      int minutes = endTime.get(Calendar.MINUTE)
     - startTime.get(Calendar.MINUTE);
      int seconds = endTime.get(Calendar.SECOND)
     - startTime.get(Calendar.SECOND);
      if (seconds < 0) {
        minutes--;
        seconds += 60;
      }
      int milliseconds = endTime.get(Calendar.MILLISECOND)
    - startTime.get(Calendar.MILLISECOND);
      if (milliseconds < 0) {
        seconds--;
        milliseconds += 1000;
      }
      return Integer.toString(minutes) + " minuti, " + seconds + " secondi, "  + milliseconds + " millisecondi";
    }

    public static void main(String[] args) {
      Chronometer chron = new Chronometer();
      // manda in esecuzione il thread per mezzo di Chronometer
      // dando come parametro al costruttore il numero massimo
      // raggiungibile dal counter, ricevuto a sua volta


  // come parametro di main, dalla linea di comando
      chron.run(new CounterThread(Integer.parseInt(args[0])));

      // stampa il tempo trascorso
      System.out.println(chron.getElapsedTime());
    }
  }
```

I metodi sleep() e join() sono metodi che hanno in comune la caratteristica di mettere un thread in stato di attesa (blocked). Ma mentre con sleep() l'attesa ha una durata prefissata, con join() l'attesa potrebbe protrarsi indefinitamente, o non avere addirittura termine. Alcune volte il protrarsi dell'attesa oltre un certo limite potrebbe indicare un malfunzionamento o comunque una condizione da gestire in maniera diversa che stando semplicemente ad aspettare.

In questi casi si può usare il metodo join(int millisecondi)che permette di assegnare un limite

massimo di attesa, dopo in quale il metodo ritornerà comunque, consentendo al metodo chiamante di riprendere l'esecuzione.

Sia sleep() che join() mettono in attesa il thread corrente, ma il metodo join() non è un metodo statico: viene chiamato per un oggetto specifico, che è quello di cui si attende la terminazione.

## Interruzione di un thread

Un'altra caratteristica che accomuna thread e processi, è quella di essere soggetti a interruzioni. Come si è visto, l'interruzione è legata a un evento particolare, in qualche modo eccezionale, che determina cambiamenti tali nel contesto dell'esecuzione da richiedere (o poter richiedere) una gestione particolare dell'evento, ossia l'esecuzione di un codice specifico che fa fronte all'evento occorso.

In un thread l'interruzione ha luogo quando da un altro thread viene chiamato il metodo interrupt() per quel thread; formalmente è vero che un thread potrebbe interrompere se stesso, ma la cosa avrebbe poco senso.

L'aspetto più rilevante di questo metodo è che è in grado di interrompere uno stato di attesa causato da una chiamata a sleep() o join() (il discorso vale anche per il metodo wait() di cui si parlerà in seguito).

Se si ripensa per un momento agli esempi precedenti in cui vengono usati questi metodi, si nota che le chiamate sono all'interno di un blocco try e che nel catch viene intercettata un'eccezione del tipo InterruptedException anche se in questi casi l'eccezione non viene gestita.

Questo è appunto l'effetto di una chiamata al metodo interrupt(): se il thread interrotto è in stato di attesa, viene generata un'eccezione del tipo InterruptedException e lo stato di attesa viene interrotto.

L'oggetto Thread ha così l'opportunità di gestire l'interruzione, eseguendo del codice all'interno del blocco catch. Se il blocco catch è vuoto, l'effetto dell'interruzione sarà semplicemente quello di far riprendere l'esecuzione (non appena il thread, passato nuovamente allo stato runnable, sarà mandato in esecuzione dallo scheduler) dall'istruzione successiva alla chiamata sleep() o join().

Cosa accade se invece viene interrotto un thread che non è in attesa? In questo caso viene modificata una variabile di stato del thread facendo sì che il metodo isInterrupted() restituisca true. Questo permette al thread di gestire ugualmente l'interruzione controllando (tipicamente alla fine o comunque all'interno di un ciclo) il valore restituito da questo metodo:

```
public void run() {
 while (true) {
   doMyJob();
   if (isInterrupted())
     handleInterrupt();
 }
}
```

Purtroppo il flag di interruzione non viene impostato se l'interruzione ha luogo durante uno stato di attesa, per cui un codice del genere non funzionerebbe correttamente:

```
public void run() {
 while (true) {
   doMyJob();

   try {
     sleep(100);
   } catch (InterruptedException e) {}

   if (isInterrupted())
     handleInterrupt();
 }
}
```

Infatti, se l'interruzione ha luogo durante l'esecuzione di sleep(), viene generata una eccezione, ma il metodo isInterrupted() restituisce false.

Se si vuole gestire l'interruzione indipendentemente dal momento in cui si verifica, bisogna duplicare la chiamata a handleInterrupt():

```
public void run() {
 while (true) {
   doMyJob();

   try {
     sleep(100);
   } catch (InterruptedException e) {
     handleInterrupt()
   }


   if (isInterrupted())
     handleInterrupt();
 }
}
```

Il metodo interrupt() generalmente non interrompe un blocco dovuto ad attesa di I/O. In questi casi si deve agire direttamente sugli stream per interrompere lo stato di attesa. Il metodo interrupt è stato introdotto con Java 1.1 e non funziona con Java 1.0. Inoltre spesso non è supportato dalla VM dei browser, anche di quelli che dovrebbero supportare Java 1.1. Quindi per ora è opportuno evitarne l'uso nelle applet, a meno che non si faccia uso del Java plug-in.

## Metodi deprecati

Il metodo stop(), che termina l'esecuzione di un thread, oltre a essere stato deprecato in Java 2, è sconsigliato: infatti il suo uso rischia di produrre malfunzionamenti causando una

interruzione "al buio" (cioè senza che il thread interrotto abbia il controllo delle modalità di terminazione. Per motivi analoghi sono deprecati i metodi `suspend()`, che mette il thread nello stato blocked, e `resume()` che lo sblocca, riportandolo allo stato runnable.

# La sincronizzazione dei thread

Parlando dei processi, nell'introduzione, si è detto che questi hanno spazi di memoria separati e che possono condividere e scambiare dati tra loro solo con mezzi particolari appositamente concepiti per questo scopo. Si è inoltre detto che i thread che appartengono al medesimo processo condividono automaticamente lo spazio di memoria.

Come si è visto, una classe `Thread` o una basata sul `Runnable` funzionano come normali classi, e come tali hanno accesso a tutti gli oggetti che rientrano nel loro scope.

La differenza fondamentale è che, mentre le normali classi funzionano una alla volta, ossia eseguono il loro codice in momenti differenti, i thread vengono eseguiti in parallelo; questo significa che esiste la possibilità che thread diversi accedano contemporaneamente agli stessi dati. Anche se per "contemporaneamente" si intende sempre qualcosa basato su un parallelismo simulato, vi sono casi in cui questa "simultaneità" di accesso, per quanto relativa, può causare effettivamente dei problemi.

Sorge così l'esigenza di implementare una qualche tecnica di sincronizzazione dei vari thread. Prima di spiegare nei dettagli gli aspetti legati alla sincronizzazione, si ponga attenzione a i modi in cui diversi thread condividono oggetti e dati.

## Condivisione di dati fra thread

Il caso più comune è quello di oggetti creati esternamente che vengono passati come parametri a un oggetto `Thread` o `Runnable`. Ad esempio:

```
public class PointXY {

 int x;
 int y;

 public PointXY(int x, int y) {
   this.x = x;
   this.y = y;
 }
}

public class PointThread extends Thread {

 PointXY point;

 public Run1(PointXY p) {
   point = p;
 }
```

```
  public void run() {
    // esegue operazioni con la variabile point
  }
}

public class PointRunnable implements Runnable {

  PointXY point;

  public Run2(PointXY p) {
    point = p;
  }

  public void run() {
    // esegue operazioni con la variabile point
  }
}

...

// dall'esterno si istanziano e si lanciano
// i thread relativi Run1 e Run2
PointXY point = new PointXY(10, 10);
PointThread thread1 = new PointThread(point);
PointRunnable runnable = new PointRunnable(point);
Thread thread2 = new Thread(runnable);

// start ed utilizzazione...
```

In questo modo si è permessa una condivisione di una variabile tra due thread, di cui uno funziona come sottoclasse di Thread, l'altro è collegato a una Runnable. Tutti e due gli oggetti hanno accesso alla stessa istanza dell'oggetto p di tipo PointXY.

Si sarebbero potute anche creare due istanze della stessa classe:

```
// condivisione di uno stesso oggetto point
// tra due istanze di uno stessa classe Thread
PointThread thread1 = new PointThread(point);
PointThread thread2 = new PointThread(point);
```

oppure due Thread collegati allo stesso Runnable:

```
// condivisione di uno stesso oggetto Runnable
// da parte di due Thread e di conseguenza
// condivisione dello stesso oggetto point
PointRunnable runnable = new PointRunnable(point);
Thread thread1 = new Thread(runnable);
Thread thread2 = new Thread(runnable);
```

In questi due casi i thread non solo condividono l'oggetto point, ma eseguono anche lo stesso codice in maniera indipendente ed eventualmente con differenti modalità. Nell'ultimo caso si è in presenza di una sola istanza di un oggetto Runnable, a cui si passa l'oggetto point, che viene "agganciato" a due thread differenti, mentre nel primo caso si creano due istanze di una sottoclasse di Thread, a cui si passa lo stesso oggetto point.

## Competizione fra thread

Dopo aver accennato ad una delle configurazioni tipiche di accesso concorrente ad aree di memoria, si può passare a considerare quali siano i potenziali problemi derivanti dalla condivisione dei dati e dall'accesso parallelo a questi dati.

Si consideri ad esempio l'interfaccia Value che funge da contenitore (wrapper) di un valore intero:

```
public interface Value {
 public abstract int get();
 public abstract void set(int i);
 public abstract void increment();
}
```

La classe IntValue implementa l'interfaccia di cui sopra fornendo una gestione del valore contenuto come intero

```
public class IntValue implements Value {
 int value = 0;

 public int get() {
    return value;
 }

 public void set(int i) {
    value = i;
 }

 public void increment() {
    value++;
 }
}
```

La classe StringValue invece fornisce una gestione del valore come stringa

```
public class StringValue implements Value {
 String value = "0";

 public int get() {
```

```
        return Integer.parseInt(value);
    }

    public void set(int i) {
        value = Integer.toString(i);
    }

    public void increment() {
        int i = get();
        i++;
        set(i);
    }
}
```

Infine il thread permette di incrementare il valore contenuto in un generico wrapper, che viene passato al costruttore come interfaccia generica

```
public class ValueIncrementer extends Thread {
    Value value;
    int increment;

    public ValueIncrementer(Value value, int increment) {
        this.value = value;
        this.increment = increment;

}

    public void run() {
        for (int i = 0; i < increment; i++)
            value.increment();
    }

    public static void main(String[] args) {
        // crea un IntValue
        IntValue intValue = new IntValue();
        // crea due IntIncrementer a cui passa lo stesso IntValue
        // e lo stesso valore di incremento pari a 100000
        ValueIncrementer intIncrementer1  = new ValueIncrementer(intValue, 100000);
        ValueIncrementer intIncrementer2  = new ValueIncrementer(intValue, 100000);
        // ripete i passi precedenti
        // questa volta con un oggetto StringValue
        StringValue stringValue = new StringValue();
        ValueIncrementer stringIncrementer1  = new ValueIncrementer(stringValue, 100000);
        ValueIncrementer stringIncrementer2  = new ValueIncrementer(stringValue, 100000);

        try {
```

```
      // fa partire insieme i due thread che
      // incrementano lo IntValue
      intIncrementer1.start();
      intIncrementer2.start();
      // attende che i due thread terminino l'esecuzione
      intIncrementer1.join();
      intIncrementer2.join();
      // stampa il valore
      System.out.println("int value: " + intValue.get());

      // ripete i passi precedenti
      // questa volta con lo StringValue
      stringIncrementer1.start();
      stringIncrementer2.start();
      stringIncrementer1.join();
      stringIncrementer2.join();
      System.out.println("string value: " + stringValue.get());
    } catch (InterruptedException e) {}
  }
}
```

Tralasciando le considerazioni legate al modo in cui si effettua l'incremento, mandando in esecuzione l'esempio (impiegando una VM che utilizza il time-slicing), si può notare che mentre il valore per l'oggetto di tipo IntValue è quello che ci si aspetta, dovuto all'incremento di 100000 effettuato da due thread distinti, il valore dell'oggetto StringValue è inferiore, e varia da esecuzione a esecuzione.

Per capire cosa sia successo si esamini il codice del metodo increment() delle due classi IntValue e StringValue. Nella classe IntValue si ha

```
public void increment() {
  value++;
}
```

ovvero il metodo compie una semplice operazione di incremento di una variabile di tipo int. Invece, nella classe StringValue si trova

```
public void increment() {
  int i = get();
  i++;
  set(i);
}
```

Qui siamo in presenza di un algoritmo che, per quanto semplice, è formato da diverse istruzioni; a loro volta i metodi get() eset() chiamano metodi della classe Integer per convertire la stringa in int e viceversa, metodi che compiono operazioni di una certa complessità, ossia eseguono diverse istruzioni, ma possiamo anche prescindere da quest'ultima osservazione. Quello

che conta è che si tratta comunque di un'operazione complessa, divisa in più passi successivi. È questa complessità dell'operazione ciò che causa il problema. Infatti, se i due thread funzionano in parallelo in un sistema gestito con il time-slicing, è possibile che il passaggio da un thread all'altro avvenga durante l'esecuzione del metodo increment().

In questo caso, dato che i thread non fanno altro che eseguire increment() in un ciclo, e hanno la stessa priorità, le probabilità sono piuttosto alte. Di conseguenza, è possibile che il processore segua una sequenza di esecuzione come questa (per semplicità le chiamate get() e set() saranno ipotizzate istruzioni semplici):

1. Il primo thread esegue l'istruzione int i = get();
   Supponendo che il valore sia 100, questo valore viene assegnato alla variabile locale i

2. Il secondo thread esegue l'istruzione int i = get();
   Il valore è sempre 100, e viene assegnato all'altra variabile locale i (che è diversa per ciascun thread)

3. Il primo thread esegue l'istruzione i++;

4. Il valore della variabile locale diventa 101
   Il primo thread esegue l'istruzione set(i);
   La variabile di classe (condivisa fra i thread) diventa 101

5. Il secondo thread esegue l'istruzione i++;
   Il valore della variabile locale passa da 100 a 101

6. Il secondo thread esegue l'istruzione set(i);
   Alla variabile di classe viene assegnato il valore 101, che è lo stesso che già aveva, in seguito all'azione del primo thread.

Quindi, come risultato complessivo si ottiene un incremento di 1 e non di 2, come se uno dei due thread non avesse fatto nulla. Questa situazione di interferenza tra thread è quella che viene generalmente chiamata *race condition*.

Si noti come nel caso degli oggetti IntValue non si sia verificata nessuna alterazione: infatti il metodo increment() di questa classe compie una sola operazione non complessa, l'incremento della variabile interna, ossia un'istruzione che non può essere interrotta nel mezzo da un cambio di contesto. Le operazioni di questo tipo sono chiamate *atomiche*.

Alcune operazioni in apparenza atomiche, possono in realtà non esserlo: infatti se una variabile di tipo int è un rappresentata da un'area indivisibile di 32 bit, e quindi un'operazione di scrittura viene eseguita in una sola operazione non interrompibile, altrettanto non vale per un long, che occupa 64 bit di memoria, e in certi casi viene scritto o letto in due blocchi di 32 bit con due operazioni distinte. Se fra una operazione e la successiva si interrompe il thread, ci può

essere un'alterazione non voluta del valore della variabile. Si tenga presente che la frammentazione di una operazione di scrittura in sottoperazioni avviene a basso livello in maniera non visibile dal programma Java. Una race condition (condizione di competizione) è la situazione che si verifica quando due o più thread eseguono contemporaneamente operazioni di cui almeno una non atomica sugli stessi dati; l'ordine con il quale i vari passi di cui le operazioni sono composte vengono eseguiti dai diversi thread può portare ad alterazioni del risultato dell'intera operazione per uno o più thread.

## Lock e sincronizzazione

Per risolvere una simile situazione è necessario che l'operazione complessa sia effettuata per intero da un thread alla volta, e che non venga interrotta da istruzioni relative alla stessa operazione eseguite da un altro thread. Per ottenere questo risultato generalmente si ricorre all'utilizzo dei cosiddetti lock.

Il lock può essere paragonato alla chiave di una toilette: alla toilette accede una sola persona alla volta e una volta entrata chiude la porta a chiave, dato che, anche in questo caso, sia pure per motivi differenti rispetto al caso dei thread, la condivisione della risorsa potrebbe portare a risultati indesiderati. Le altre persone che vogliono entrare, trovano la porta chiusa e devono pertanto attendere l'uscita dell'utente corrente della toilette.

Il lock può essere pensato come una semplice variabile booleana, visibile da tutti i thread. Ogni volta che un thread esegue un codice protetto da un lock, la variabile viene impostata a `true`, per indicare che il codice è già in esecuzione, e si dirà che il thread ha acquisito il lock su quel codice.

Il meccanismo di "schedulazione" si fa carico di garantire che, fin quando un thread è in possesso di un lock su un certo codice, nessun altro thread vi acceda. Eventuali thread che chiedono l'accesso al codice vengono così messi in attesa finché il thread corrente non ha rilasciato il lock.

In Java, però, tutto questo avviene "dietro le quinte", dal momento che il programmatore non usa direttamente dei lock, ma ricorre invece alla keyword `synchronized`. Il meccanismo è estremamente semplice.

Si riconsideri l'esempio di cui sopra facendo una piccola ma importante modifica al metodo `increment()`

```
public synchronized void increment() {
    int i = get();
    i++;
    set(i);
}
```

Se si prova adesso a eseguire l'applicazione, si potrà vedere che il risultato è corretto, per entrambe le classi `IntValue` e `StringValue`.

Quando un thread esegue un metodo synchronized, acquisisce il lock prima di eseguire le istruzioni, e lo rilascia al termine dell'esecuzione.

Ma acquisisce il lock su che cosa? La risposta è: acquisisce il lock sull'oggetto stesso. Più precisamente quando un oggetto è sottoposto a lock, nessuno dei suoi metodi sincronizzati è eseguibile se non dal thread che detiene il lock (cosa molto importante al fine di evitare deadlock).

Per capire meglio cosa questo significhi in pratica, si consideri una nuova classe ValueIncrementer2 identica alla classe ValueIncrementer, ma con una piccola modifica nel metodo run():

```
public void run() {
   for (int i = 0; i < increment; i++)
      value.set(value.get() + 1);
}
```

In questo caso l'incremento non è più ottenuto con una chiamata al metodo increment(), ma chiamando direttamente i metodi get() e set().

Se si prova ad eseguire contemporaneamente un oggetto ValueIncrementer e un oggetto ValueIncrementer2, nonostante la sincronizzazione del metodo increment() della classe StringValue, si otterrà una race condition con forti probabilità di funzionamento anomalo.

Il motivo di questa incomprensibile stranezza risiede nel fatto che increment() è l'unico metodo sincronizzato: ciò implica non solo che sia l'unico ad acquisire il lock, ma anche che sia l'unico a rispettarlo. In sostanza il lock non ha alcun effetto sui metodi non sincronizzati, in particolare sui metodi get() e set(), che quindi possono essere eseguiti in parallelo a increment() e causare i problemi che abbiamo visto.

Per evitare questi problemi, si devono definire come synchronized anche i metodi get() e set(). In tal modo, poiché il lock è sull'oggetto, sarà impossibile mandare contemporaneamente in esecuzione due metodi sincronizzati dello stesso oggetto; nel nostro caso i metodi increment(), get() e set() non potranno essere eseguiti in parallelo sullo stesso oggetto StringValue, ma dovranno attendere ognuno la fine dell'esecuzione dell'altro su un altro thread. Infine si tenga presente che una variabile non può essere direttamente sottoposta a lock, dato che si possono sincronizzare solo i metodi.

Quindi per permettere realmente la sincronizzazione sull'accesso concorrente a una variabile, oltre a definire sincronizzati tutti i metodi di gestione di tale variabile, si dovrà impedire l'accesso diretto per mezzo di una istruzione del tipo

*oggetto.variabile = valore*

Quindi tutte le variabili passibili di accesso condiviso devono essere protette e ad esse si deve accedere esclusivamente con metodi sincronizzati pubblici.

Un oggetto si dice thread-safe quando è protetto da malfunzionamenti causati da race condition e quindi è correttamente eseguibile anche contemporaneamente da thread differenti. Lo stesso termine può essere riferito anche a singoli metodi o a intere librerie.

## Visibilità del lock

L'uso di synchronized fino ad ora è stato applicato a un intero metodo. Esiste anche la possibilità di circoscrivere l'ambito della sincronizzazione a un blocco di codice, ottenendo così un blocco sincronizzato. Ecco un'altra versione del metodo increment() di StringValue, che esemplifica questa modalità:

```
public void increment() {
  synchronized (this) {
    int i = get();
    i++;
    set(i);
  }
}
```

In questo caso le due versioni del metodo sono esattamente equivalenti, dato che il lock viene acquisito all'inizio del metodo e rilasciato alla fine.

Può capitare però che solo una porzione di codice all'interno di un metodo necessiti di sincronizzazione. In questi casi può essere opportuno usare un blocco sincronizzato piuttosto che un metodo sincronizzato, restringendo lo scope del lock.

Per visibilità di un lock (*scope* del lock) si intende il suo ambito di durata, corrispondente alla sequenza di istruzioni che viene eseguita tra il momento in cui il lock viene acquisito e quello in cui viene rilasciato.

Utilizzando il blocco sincronizzato si deve anche specificare l'oggetto di cui vogliamo acquisire il lock. Nell'esempio precedente l'oggetto è lo stesso di cui si sta eseguendo il metodo, e quindi è indicato con this, ma potrebbe essere anche un altro.

Questo significa che un lock su un oggetto può aver effetto anche su codice di altri oggetti, anche di classi differenti. Quindi, correggendo un'affermazione precedentemente fatta, dal contenuto ancora impreciso, si può dire che quando un oggetto è sottoposto a lock, nessuna area sincronizzata — intendendo sia blocchi che metodi — che richieda il lock per quel determinato oggetto è eseguibile se non dal thread che detiene il lock.

Tra brevissimo sarà preso in esame un esempio di uso di un blocco sincronizzato per un oggetto diverso da this. Ma quali sono i criteri in base ai quali scegliere lo scope appropriato? Bisogna naturalmente valutare caso per caso tenendo presente i diversi aspetti a favore e contro.

Da una parte, uno scope più esteso del necessario può causare inutili ritardi nell'esecuzione di altri thread, e in casi particolari può anche portare a una situazione di stallo, detta deadlock, di cui si dirà tra poco.

Dall'altra, acquisire e rilasciare un lock è un'operazione che consuma delle risorse e quindi, se si verifica troppo di frequente, rischia di influire negativamente sull'efficienza del programma. Inoltre, come vedremo, anche l'acquisizione di troppi lock può portare al verificarsi di deadlock.

Lo scope di un'area sincronizzata in Java non può estendersi al di là di un singolo metodo. Nel caso servano lock di scope più estesi (che vengano acquisiti in un metodo e rilasciati in un'altro, eventualmente di un'altro oggetto) occorre ricorrere a lock implementati *ad hoc* (ad esempio una classe Lock) la cui trattazione esula dagli scopi di questo capitolo.

## Deadlock

Si supponga di scrivere una classe FileUtility che fornisca una serie di funzioni di utilità per il file system. Una delle funzioni è quella di eliminare da una directory tutti i files la cui data è precedente a una certa data fissata dall'utente, oppure esistenti da più di un certo numero di giorni. Un'altra funzione è di comprimere i files di una certa directory.

Si supponga di aver creato due classi:

la classe File, che tra l'altro contiene un metodo isOlder(Date d) che controlla se il file è antecedente a una certa data, e un metodo compress() che comprime il file;

la classe Directory, che contiene tra gli altri un metodo removeFile(File f), e dei metodi firstFile() e nextFile() utilizzabili per iterare sui files della directory, che sono mantenuti come una collezione di oggetti File all'interno dell'oggetto Directory.

La classe FileUtility, da parte sua, contiene removeOldFiles(Directory dir, Date date), un metodo che elimina i files "vecchi", e compressFiles(Directory dir), un metodo che comprime tutti i file di una directory.

Questa potrebbe essere una implementazione del metodo removeOldFiles:

```
public void removeOldFiles (Directory dir, Date date) {
 for (File file = dir.firstFile(); file != null; dir.nextFile()) {
synchronized (file) {
      if (file.isOlder(date)) {


      synchronized (dir) {
            dir.removeFile(file);
         }
      }
    }
 }
}
```

Questo è un tipico esempio di uso del blocco sincronizzato su un oggetto diverso da this: quello che serve è un lock sul file, per evitare che altri thread possano agire contemporaneamente sullo stesso file. Se il file risulta "vecchio" si utilizza il metodo removeFile() dell'oggetto Directory, ed anche in questo caso si deve ottenere il lock su tale oggetto, per evitare interventi contemporanei sulla stessa directory, che potrebbero interferire con l'operazione di cancellazione.

Questa potrebbe essere una possibile implementazione del metodo compressFiles():

```
public void compressFiles (Directory dir) {
 synchronized (dir) {
   for (File file = dir.firstFile(); file != null; dir.nextFile()) {
     synchronized (file) {
       file.compress();
     }
   }
 }
}
```

Anche in questa circostanza il thread deve acquisire i lock sull'oggetto Directory e sull'oggetto File per evitare interferenze potenzialmente dannose.

Si ipotizzi adesso che i due thread siano in esecuzione contemporaneamente e che si verifichi una sequenza di esecuzione come questa:

1. Il primo thread chiama il metodo compressFiles() per un certo oggetto Directory, acquisendone il lock;

2. Il secondo thread chiama il metodo removeOldFiles() per lo stesso oggetto Directory, verificando che il primo file è vecchio, e acquisisce il lock per il primo oggetto File;

3. Il secondo thread, per procedere alla rimozione del file, tenta di acquisire il lock sull'oggetto Directory, lo trova già occupato e si mette in attesa;

4. Il primo thread tenta di acquisire il lock per il primo oggetto File, lo trova occupato e si mette in attesa.

A questo punto i thread si trovano in una situazione di stallo, in cui ognuno aspetta l'altro, ma l'attesa non avrà mai termine. Si è verificato un deadlock.

Un deadlock è una situazione in cui due o più thread (o processi) si trovano in attesa l'uno dell'altro, in modo tale che gli eventi attesi non potranno mai verificarsi.

Per quanto riguarda la prevenzione dell'insorgenza di deadlock, non ci sono mezzi particolari messi a disposizione dal linguaggio, né regole generali e precise da seguire.

Si tratta di esaminare con attenzione le possibili interazioni fra thread e tenerne conto nell'implementazione delle classi. Ci sono naturalmente dei casi tipici, il cui esame va però al di là degli obiettivi di questo capitolo.

## Class lock e sincronizzazione di metodi statici

La keyword synchronized può essere usata anche per metodi statici, ad esempio per sincronizzare l'accesso a variabili statiche della classe. In questo caso quello che viene acquisito è il lock della classe, anziché di un determinato oggetto di quella classe.

In realtà il lock si riferisce sempre a un oggetto, e precisamente all'oggetto Class che rappresenta quella classe. Quindi è possibile acquisire un lock della classe anche da un blocco sincronizzato, specificando l'oggetto Class:

```
public void someMethod() {
 synchronized (someObject.class) {
   doSomething();
 }
}
```

oppure:

```
public void someMethod() {
 synchronized (Class.forName("SomeClass")) {
   doSomething();
 }
}
```

# Comunicazione fra thread

Dato che la programmazione per thread permette l'esecuzione contemporanea di più flussi di esecuzione autonomi fra loro, sorge abbastanza spontanea l'esigenza di mettere in comunicazione fra loro tali flussi in modo da realizzare qualche tipo di lavoro collaborativo. Il modo più semplice per ottenere la comunicazione fra thread è la condivisione diretta di dati, attraverso codice sincronizzato, come visto in precedenza. Ma ci sono situazioni in cui questo sistema non è sufficiente.

## Condivisione di dati

Si consideri questo semplice esempio basato su le due classi Transmitter e Receiver, utilizzabili su thread differenti per lo scambio di dati:

```
public class Transmitter extends Thread {
 Vector data;

 public Transmitter(Vector v) {
   data = v;
 }

 public void transmit(Object obj) {
   synchronized (data) {
     data.add(obj);
   }
 }
```

```
  public void run() {
    int sleepTime = 50;
    transmit("Ora trasmetto 10 numeri");
    try {
      if (!isInterrupted()) {
        sleep(1000);
        for (int i = 1; i <= 10; i++) {
          transmit(new Integer(i * 3));
          if (isInterrupted())
            break;
          sleep(sleepTime * i);
        }
      }
    } catch (InterruptedException e) {}
    transmit("Fine della trasmissione");
  }
}
```

La classe Transmitter implementa un semplice meccanismo di condivisione dei dati attraverso un oggetto Vector, che viene passato come argomento del costruttore. Il metodo transmit() non fa altro che aggiungere un elemento al Vector, dopo aver acquisito un lock sul Vector stesso.

Questa operazione ha un reale effetto perché la classe Vector è una classe thread-safe, ossia è stata implementata usando dove necessario dei blocchi o dei metodi sincronizzati.

Il metodo run() trasmette un messaggio iniziale, attende un secondo, poi trasmette una sequenza di 10 numeri a intervalli di tempo crescenti, infine trasmette un messaggio finale.

In questo metodo viene anche esemplificata una gestione delle interruzioni: i messaggi iniziale e finale vengono comunque trasmessi; in caso di interruzione durante la trasmissione viene conclusa la trasmissione in corso, poi si esce dal ciclo; se l'interruzione arriva durante una chiamata a sleep(), questa causa un salto al blocco catch, vuoto, con un risultato equivalente.

```
public class Receiver extends Thread {
  Vector data;

  public Receiver(Vector v) {
    data = v;
  }

  public Object receive() {
    Object obj;
    synchronized (data) {
      if (data.size() == 0)
        obj = null;
      else {
        obj = data.elementAt(0);
```

```
            data.removeElementAt(0);
         }
      }
      return obj;
   }

   public void run() {
      Object obj;
      while (!isInterrupted()) {
         while ((obj = receive()) == null) {
            try {
               sleep(1000);
            } catch (InterruptedException e) {
               return;
            }
         }
         System.out.println(obj.toString());
      }
   }
}
```

La classe Receiver riceve anch'essa un Vector come argomento del costruttore, e in tal modo ha la possibilità di condividere i dati con un Transmitter.

Il metodo receive() restituisce null se non trova dati; altrimenti restituisce il dato dopo averlo rimosso dal Vector.

Il metodo run() esegue un ciclo che può essere terminato solo da una chiamata a interrupt().

Anche in questo caso viene gestito sia la possibilità di interruzione in stato di attesa con una InterruptedException, sia quella di interruzione durante l'esecuzione.

Ad ogni ciclo si prova a ricevere un dato: se la ricezione ha luogo, stampa il dato sotto forma di stringa, altrimenti attende un secondo e riprende il ciclo.

```
public class ThreadCommunication{
   public static void main(String[] args) {
      Vector vector = new Vector();
      Transmitter transmitter = new Transmitter(vector);
      Receiver receiver = new Receiver(vector);
      transmitter.start();
      receiver.start();
      try {
         transmitter.join();
         Thread.sleep(2000);
      } catch (InterruptedException e) {}
      receiver.interrupt();
   }
}
```

La classe ThreadCommunication contiene soltanto un main() che mostra il funzionamento delle classi Transmitter e Receiver: in tale metodo viene creato un oggetto di tipo Vector e lo passa ai due oggetti Transmitter e Receiver, che poi provvede a far partire.

Quando il Transmitter ha terminato la sua esecuzione, attende 2 secondi per dare al Receiver il tempo di ricevere gli ultimi dati; successivamente termina il Receiver con una chiamata al metodo interrupt().

In questo caso il metodo interrupt() è usato per terminare il thread: questo è un sistema che può essere usato nei casi in cui non ci sia la necessità gestire le interruzioni diversamente, senza terminare il thread, ma eseguendo un determinato codice. La differenza, rispetto all'uso di un flag di stop, è che il thread termina immediatamente anche se si trova in stato di attesa.

## Utilizzo dei metodi wait() e notify()

Per migliorare la sincronizzazione fra i thread, si può ricorrere all'utilizzo dei metodi wait() e notify().

Si tratta di metodi appartenenti alla classe Object e non alla classe Thread, per cui possono essere utilizzati per qualunque oggetto. Il funzionamento è semplice: se un thread esegue una chiamata al metodo wait() di un determinato oggetto, il thread rimarrà in stato di attesa fino a che un altro thread non chiamerà il metodo notify() di quello stesso oggetto.

Con poche variazioni al codice, si possono modificare le classi appena viste in modo che utilizzino wait() e notify().

Per prima cosa si può si può modificare il metodo transmit() della classe Transmitter in modo che sia effettuata una chiamata al metodo notify() per segnalare l'avvenuta trasmissione:

```
public void transmit(Object obj) {
 synchronized (data) {
    data.add(obj);
    data.notify();
 }
}
```

Nella classe Receiver invece il metodo run() diventa

```
public void run() {
 Object obj;
 while (!isInterrupted()) {
    synchronized (data) {
       while ((obj = receive()) == null) {
          try {
             data.wait();
          } catch (InterruptedException e) {
             return;
          }
```

```
            }
        }
        System.out.println(obj.toString());
    }
}
```

Come si può notare al posto della chiamata a sleep() si effettua una chiamata a wait(): in tal modo l'attesa si interromperà subito dopo la trasmissione, segnalata dalla chiamata notify(). Per il resto il funzionamento resta uguale.

Si può notare però un'altra differenza: all'interno del ciclo è stato inseritoun blocco sincronizzato, pur essendo il metodo receive() già sincronizzato.

Il motivo è che utilizzando direttamente la wait() è necessario prima ottenere il lock sull'oggetto, altrimenti si causa un'eccezione del tipo IllegalMonitorStateException, che darà il poco intuitivo messaggio current thread not owner, che sta a significare che il thread non detiene il lock sull'oggetto.

Più precisamente questo significa che, essendo la wait() un metodo della classe Object e non della Thread, è possibile invocare una wait() su tutti gli oggetti Java, non solo sui thread, anche se l'invocazione può avvenire solo se preventivamente si è acquisito il lock relativo. Discorso del tutto analogo per il metodo notify().

Questa è la tipica sequenza di eventi per la creazione di una sincronizzazione fra due thread; supponendo che il thread thread1 chiami wait() e il thread thread2 chiami notify():

1. il thread1 chiama wait() dopo aver acquisito il lock sull'oggetto; wait() per prima cosa rilascia il lock, poi mette il thread in attesa;

2. il thread2 a questo punto può acquisire il lock ed eseguire il blocco sincronizzato da cui chiama notify() dopodiché il lock sarà rilasciato…

3. …da thread1; wait(), ricevuta la notifica riacquisisce il lock ed esce; successivamente l'esecuzione del codice potrà continuare.

Tipicamente questo tipo di comunicazione si usa per aspettare/notificare il verificarsi di una certa condizione.

Nell'esempio appena visto la condizione è la presenza di dati ricevuti; in mancanza del meccanismo di sincronizzazione descritto sopra, e permettendo l'uso di wait() e notify() al di fuori di aree sincronizzate, si potrebbe verificare una sequenza di questo tipo:

1. il Receiver controlla se ci sono dati ricevuti; non ne trova;

2. il Transmitter trasmette un dato,

3. il Transmitter chiama notify(), ma la notifica non arriva a destinazione, dal momento che non c'è ancora nessun thread in attesa;

4. il Receiver si mette in attesa, ma ormai la notifica è andata persa.

È importante osservare che se un thread acquisisce il lock su un determinato oggetto, solo esso potrà eseguire l'operazione di rilascio e, finché non effettuerà tale operazione (uscendo dal blocco sincronizzato), il lock risulterà sempre occupato.

Questo fatto ha un'importante conseguenza: il thread messo in stato di attesa, che restituisce temporaneamente il lock, potrà essere riattivato solo se, dopo la chiamata a notify(), viene ad esso restituito il lock sull'oggetto che originariamente aveva acquisito. Non è detto quindi che un thread riprenda immediatamente la sua esecuzione dopo una chiamata a notify(), ma può trascorrere un periodo di tempo non precisato.

Ad esempio, se si scrive

```
synchronized (object){
 doSomething();
 object.notify();
 doSomethingElse();
}
```

fino a che non viene terminata l'esecuzione di doSomethingElse(), il thread che ha chiamato wait() non può riprendere l'esecuzione, anche se ne è stata richiesta la riattivazione con notify().

Il metodo wait() esiste anche nella versione wait(int millisecondi): in questo caso viene specificato un timeout scaduto il quale lo stato di attesa termina anche se non è stata ricevuta alcuna notifica.

Questo metodo può essere usato al posto di sleep() quando si vuole bloccare momentaneamente il thread rilasciando contemporaneamente il lock acquisito.

## Il metodo notifyAll()

Se i thread in accesso concorrente sono più di uno, e tutti in attesa a causa di una wait(), allora una chiamata alla notify() avvertirà uno solo dei thread in attesa, senza la possibilità di sapere quale. In situazioni del genere il metodo notifyAll(), permette di eseguire la notifica nei confronti di tutti i thread in attesa.

Nel caso in cui si desideri che sia solo un particolare thread tra quelli in attesa a riprendere l'esecuzione, si deve implementare del codice *ad hoc* che gestisca la situazione, dato che il linguaggio Java non mette a disposizione nessuno costrutto particolare.

# Deamon thread

I thread in Java possono essere di due tipi: *user thread* o *deamon thread*. Il termine *deamon* è stato usato inizialmente per designare un certo tipo di processi nei sistemi operativi multitasking (in particolare in ambiente Unix), ossia dei processi "invisibili" che girano "in background" e svolgono dietro le quinte dei servizi di carattere generale. In genere questi processi restano in

esecuzione per tutta la sessione del sistema. Il termine "demone" è stato usato probabilmente in analogia con "fantasma" a simboleggiare invisibilità e onnipresenza.

I deamon thread in Java sono qualcosa di molto simile ai processi deamon: sono infatti thread che spesso restano in esecuzione per tutta la durata di una sessione della virtual machine, ma soprattutto sono thread che si suppone che svolgano dei servizi per gli user thread, e che questa sia l'unica ragione della loro esistenza. In effetti l'unica differenza tra uno user thread e un deamon thread è che la virtual machine termina la sua esecuzione quando termina l'ultimo user thread, indipendentemente dal fatto che ci siano o meno in esecuzione dei deamon thread.

Spesso i deamon thread sono thread creati e mandati automaticamente in esecuzione dalla stessa virtual machine: un caso tipico è quello del garbage collector, che si occupa periodicamente di liberare la memoria allocata per oggetti non più in uso.

Ma un deamon thread può essere anche creato dall'utente, cioè dal programmatore: a tale scopo esiste il metodo `setDeamon(boolean value)` che permette di rendere deamon uno user thread o, viceversa, user un deamon thread.

Per default un thread, quando viene creato assume lo stato del thread "padre" da cui è stato creato. Con `setDeamon()` è possibile modificare questo stato, ma soltanto prima di mandare in esecuzione il thread con `start()`.

Una chiamata durante l'esecuzione causerà un'eccezione. Per conoscere il deamon state di un thread si può usare il metodo `isDeamon()`.

Se si creano dei thread di tipo deamon, occorre sempre tener presente che non devono svolgere delle operazioni che possano protrarsi oltre la durata di esecuzione degli user thread per cui svolgono i loro servizi. Questo rischierebbe di interrompere a metà queste operazioni, perché la Virtual Machine potrebbe terminare per mancanza di user thread in esecuzione.

# I gruppi di thread

Ogni thread in Java appartiene a un gruppo; per default il gruppo di appartenenza è quello del thread padre. La virtual machine genera automaticamente dei thread groups, di cui almeno uno è destinato ai thread creati dalle applicazioni; questo sarà il gruppo di default per i thread creati da un'applicazione. Ogni applicazione può anche creare i suoi thread group, e assegnare i thread a questi gruppi.

I thread group sono organizzati secondo una struttura gerarchica ad albero: ciascun thread appartiene a un gruppo, il quale può appartenere a un altro gruppo; per questo ogni gruppo può contenere sia thread che gruppi di thread. La radice di quest'albero è rappresentata dal system thread group.

Per creare un thread group si deve creare un oggetto della classe `ThreadGroup` usando uno dei due costruttori:

```
ThreadGroup(String name)
ThreadGroup(ThreadGroup parent, String name);
```

Come per i thread, se non viene specificato un thread group di appartenenza, il gruppo di appartenenza sarà quello del thread da cui è stato creato.

Per assegnare un thread a un gruppo si usa uno dei tre costruttori:

```
Thread(ThreadGroup group, String name),
Thread(ThreadGroup group, Runnable target),
Thread(ThreadGroup group, Runnable target, String name).
```

Una volta creato il thread come membro di un certo gruppo, non è possibile farlo passare ad un altro gruppo o toglierlo dal gruppo. Il thread sarà rimosso automaticamente dal gruppo una volta terminata la sua esecuzione.

Le funzionalità relative ai gruppi di thread si possono suddividere in quattro categorie: funzionalità di informazione, manipolazione collettiva dei thread appartenenti a un gruppo, funzioni relative alla priorità e funzioni legate alla sicurezza. Nei paragrafi successivi si analizzano tali funzionalità.

## Informazioni sui thread e sui gruppi

Ci sono diversi metodi appartenenti alla classe ThreadGroup e alla classe Thread che forniscono informazioni sui thread e sui gruppi di thread.

Ci sono metodi che ci informano su quanti e quali sono i thread e i gruppi attualmente esistenti nella VM.

Il più importante è il metodo enumerate(), che fornisce la lista dei thread o dei thread group attivi, effettuando opzionalmente una ricorsione in tutti i sottogruppi.

Vi sono poi metodi che informano sui "rapporti di parentela" come getThreadGroup() della classe Thread o getParent() della ThreadGroup che permettono di conoscere il gruppo di appartenenza di un thread o di un gruppo.

## Thread group e priorità

Con il metodo setMaxPriority(int priorità) è possibile assegnare a un gruppo una priorità massima. Se si tenta di assegnare a un thread del gruppo (o di sottogruppi) una priorità maggiore, questa viene automaticamente ridotta alla priorità massima del gruppo, senza che venga segnalato alcun errore.

La priorità massima può essere soltanto diminuita, e non aumentata. La priorità dei thread appartenenti al gruppo non viene in realtà modificata se si abbassa la priorità massima del gruppo, anche se è più alta di tale limite. La limitazione diviene attiva solo quando viene creato un nuovo thread o viene modificata la priorità di un thread con il metodo setPriority(). In questo caso non sarà possibile superare la priorità massima del gruppo.

Il valore della priorità di un gruppo può essere ottenuto con una chiamata al metodo getPriority().

## Thread group e sicurezza

Le funzionalità più interessanti e più importanti legate ai thread group sono quelle relative alla sicurezza. Con i gruppi di thread è possibile consentire o interdire in maniera selettiva a interi gruppi di thread l'accesso ad altri thread e gruppi di thread.

Questa funzionalità è legata al funzionamento della classe java.lang.SecurityManager, la quale gestisce diverse funzioni legate alla sicurezza, tra cui alcune relative ai thread. In realtà queste funzioni sono riferite ai thread group, ma anche ai singoli thread.

Tuttavia i thread group assumono una particolare rilevanza perché consentono di discriminare l'accesso tra i thread sulla base dell'appartenenza ai gruppi, quindi accrescendo notevolmente le possibilità di organizzare i criteri di accesso secondo regole ben precise.

Il SecurityManager è quello che, ad esempio, si occupa di garantire che le Applet non possano accedere a determinate risorse del sistema. In questo caso si tratta di un SecurityManager fornito e gestito dal browser e dalla virtual machine del browser, a cui l'utente non ha accesso.

Ma per le applicazioni l'utente può invece creare e installare dei suoi SecurityManager. Prima di Java 2, le applicazioni non avevano nessun SecurityManager di default, quindi c'era solo la possibilità di usare dei SecurityManager creati dall'utente.

In Java 2 esiste anche un SecurityManager di default per le applicazioni, che può essere fatto partire con una opzione della VM, e configurato attraverso una serie di files di configurazione.

Tralasciando una descrizione complessiva del SecurityManager, le funzioni relative ai thread si basano in pratica su due soli metodi, o più precisamente su due versioni dello stesso metodo checkAccess(): questo infatti può prendere come parametro un oggetto Thread oppure un oggetto ThreadGroup.

A loro volta, le classi Thread e ThreadGroup contengono ciascuna un metodo checkAccess() che chiama i rispettivi metodi del SecurityManager.

Questo metodo viene chiamato da tutti i metodi della classe Thread e della classe ThreadGroup che determinano un qualsiasi cambiamento di stato nell'oggetto per cui vengono chiamati, per accertare che il thread corrente abbia il permesso di manipolare il thread in questione (che può essere lo stesso thread corrente o un altro thread).

Se le condizioni di accesso non sussistono, checkAccess() lancia una SecurityException che in genere viene semplicemente rilanciata dal metodo chiamante.

Le condizioni di accesso sono quindi stabilite dai metodi del SecurityManager, che può a tale scopo utilizzare tutte le informazioni che è in grado di conoscere sugli oggetti Thread o ThreadGroup di cui deve fare il check. Ad esempio può vietare l'accesso se il thread corrente e il thread in esame non appartengono allo stesso gruppo, o a seconda delle rispettive priorità, ecc.

I metodi che chiamano checkAccess() prima di compiere le loro operazioni sono:

1. nella classe Thread: Thread(), interrupt(), setPriority(), setDeamon(), setName(), più i metodi deprecati: stop(), suspend(), resume();

2. nella classe ThreadGroup: ThreadGroup(), interrupt(), setMaxPriority(), setDeamon(), destroy(), più i metodi deprecati: stop(), suspend(), resume().

Tutti questi metodi lanciano una SecurityException se il thread corrente non ha accesso al Thread o al ThreadGroup dell'oggetto this.

## La classe ThreadLocal

Questa classe è stata introdotta con Java 2. Consente di avere una variabile locale al thread, cioè ciascun thread ha una sua istanza della variabile. Il valore della variabile è ottenuto tramite i metodi get() e set() dell'oggetto ThreadLocal. L'uso tipico di quest'oggetto è come variabile privata statica di una classe in cui si vuole mantenere uno stato o un identificatore per ogni thread.

# La grafica in Java

ANDREA GINI

Uno dei problemi più grossi emersi durante la progettazione di Java fu senza dubbio la realizzazione di un toolkit grafico capace di funzionare con prestazioni di buon livello su piattaforme molto differenti tra loro. La soluzione adottata nel 1996 fu AWT, un package grafico che mappa i componenti del sistema ospite con apposite classi dette peer, scritte in gran parte in codice nativo. In pratica, ogni volta che il programmatore crea un componente AWT e lo inserisce in un'interfaccia grafica, il sistema AWT posiziona sullo schermo un oggetto grafico della piattaforma ospite, e si occupa di inoltrare ad esso tutte le chiamate a metodo effettuate sull'oggetto Java corrispondente, ricorrendo a procedure scritte in buona parte in codice nativo; nel contempo, ogni volta che l'utente manipola un elemento dell'interfaccia grafica, un'apposita routine (scritta sempre in codice nativo) crea un apposito oggetto `Event` e lo inoltra al corrispondente oggetto Java, in modo da permettere al programmatore di gestire il dialogo con il componente e le azioni dell'utente con una sintassi completamente Object Oriented e indipendente dal sistema sottostante.

A causa di questa scelta progettuale, il set di componenti grafici AWT comprende solamente quel limitato insieme di controlli grafici che costituiscono il minimo comune denominatore tra tutti i sistemi a finestre esistenti: un grosso limite rispetto alle reali esigenze dei programmatori. In secondo luogo, questa architettura presenta un grave inconveniente: i programmi grafici AWT assumono un aspetto ed un comportamento differente a seconda della JVM su cui vengono eseguite, a causa delle macroscopiche differenze implementative esistenti tra le versioni di uno stesso componente presenti nelle diverse piattaforme. Spesso le interfacce grafiche realizzate su una particolare piattaforma mostrano grossi difetti se eseguite su un sistema differente, arrivando in casi estremi a risultare inutilizzabili.

Nel 1998, con l'uscita del JDK 1.2, venne introdotto il package Swing, i cui componenti erano stati realizzati completamente in Java, ricorrendo unicamente alle primitive di disegno più semplici, tipo "traccia una linea" o "disegna un cerchio", accessibili attraverso i metodi dell'oggetto `Graphics`, un oggetto AWT utilizzato dai componenti Swing per interfacciarsi con la piattaforma ospite. Le primitive di disegno sono le stesse su tutti i sistemi grafici, e il loro utilizzo non presenta sorprese: il codice java che disegna un pulsante Swing sullo schermo di un PC produrrà lo stesso identico risultato su un Mac o su un sistema Linux. Questa architettura risolve alla radice i problemi di uniformità visuale, visto che la stessa identica libreria viene ora utilizzata, senza alcuna modifica, su qualunque JVM. Liberi dal vincolo del "minimo comune denominatore", i progettisti di Swing hanno scelto di percorrere la via opposta, creando un package ricco di componenti e funzionalità spesso non presenti nella piattaforma ospite.

## Applet e AWT

A quasi 8 anni dalla sua introduzione, Swing ha completamente rimpiazzato AWT nello sviluppo di applicazioni stand alone. Esiste tuttavia un contesto applicativo in cui l'uso di AWT risulta essere una scelta obbligatoria: la creazione di Applet Java compatibili con tutti i browser presenti sul mercato.

Un'Applet è un'applicazione Java incorporata in una pagina web, che viene scaricata assieme alla pagina ed eseguita da una JVM presente nel browser del client. Nel '96, quando Java venne introdotto sul mercato, la Netscape si accordò con la Sun Microsystems per incorporare una Java Virtual Machine nel proprio browser, allora leader di mercato. Questa decisione diede un enorme impulso alla diffusione del nuovo linguaggio, che permetteva di aggiungere alle pagine internet piccoli giochi, animazioni o vere e proprie applicazioni client. Nel '97, la Microsoft decise di adottare una politica aggressiva per conquistare la nascente Web Economy: con il lancio del browser Internet Explorer diede vita a quella che venne definita dalla stampa "La Guerra dei Browser". Verso la fine del '98, la Netscape, stremata dalla competizione, venne rilevata dal provider americano AOL, un evento che sancì la vittoria di Microsoft in una battaglia che purtroppo lasciò sul campo numerose vittime. Una delle vittime più illustri di questa competizione fu proprio Java, una tecnologia che la Microsoft intendeva ottimizzare per i propri sistemi operativi a discapito della natura multi piattaforma, vero e proprio asso nella manica del linguaggio della Sun Microsystem, Questa politica commerciale spinse la Sun Microsystems a chiamare in causa il gigante di Redmond, ottenendo una sentenza che imponeva alla Microsoft di rendere la propria Virtual Machine compatibile al 100% con le specifiche del linguaggio. La Microsoft a questo punto decise di sospendere lo sviluppo della Virtual Machine inclusa in Internet Explorer, che non venne aggiornata alle successive release del JDK. Data la posizione dominante di Microsoft nel campo dei browser, i programmatori Java sono costretti a sviluppare Applet conformi alla specifica 1.1 del linguaggio, la versione precedente all'introduzione di Swing, un fattore che ha contribuito all'arresto della diffusione di Applet nei siti web. Incidentalmente, la piattaforma Java ha finito per conquistare una posizione dominante come linguaggio per lo sviluppo di applicazioni lato Server, grazie, alle tecnologie Servlet, JSP ed EJB. Nel '97 un simile sviluppo era semplicemente impensabile, ma già in passato la tecnologia ha finito per intraprende delle strade non previste dai loro stessi creatori.

Le versioni più recenti dell'ambiente di sviluppo Java comprendono un framework per la creazione di Applet Swing, ma la loro esecuzione all'interno di un browser richiede l'installazione da parte dell'utente di un apposito plug-in da 10 MB, una circostanza che di fatto ne blocca la diffusione nel Web, a vantaggio di tecnologie alternative (Servlet, JSP, ASP e PHP) che permettono di realizzare siti Web dinamici senza problemi di compatibilità.

Queste considerazioni hanno condotto alla decisione di non trattare le tecnologie Applet ed AWT in questo manuale. Chi per ragioni professionali dovesse affrontare lo studio di tali tecnologie (esiste tuttora una grossa base installata di Applet introdotte nella seconda metà degli anni novanta in siti web tuttora attivi), non incontrerà comunque alcuna difficoltà, data l'enorme disponibilità di documentazione gratuita su web (anche in italiano), e la sostanziale somiglianza tra le API Swing ed AWT.

In questo capitolo verranno illustrati i principi della programmazione di interfacce grafiche in Java; a partire dal prossimo verranno introdotti i principali componenti grafici.

# I top level container

I top level container sono i componenti all'interno dei quali si creano le interfacce grafiche: ogni programma grafico ne possiede almeno uno, di solito un JFrame, che rappresenta la finestra principale. Ogni top level container possiede un pannello (accessibile tramite il metodo getContentPane()) all'interno del quale vanno disposti i controlli dell'interfaccia grafica. Esistono due tipi principali di finestra: JFrame e JDialog. Il primo viene solitamente usato come finestra principale per il programma, mentre il secondo serve a creare le finestre di dialogo con l'utente. Queste classi sono presenti nel package javax.swing, che deve essere importato all'inizio dell'applicazione. Data la dipendenza di Swing da alcune classi presenti nel package AWT (principalmente la classe Graphics e i LayoutManager), è quasi sempre necessario necessario importare anche il package java.awt.

È sempre possibile impostare il titolo ricorrendo al metodo setTitle(String s). Due importanti proprietà dell'oggetto sono la dimensione e la posizione, che possono essere impostate sia specificando le singole componenti sia mediante oggetti Dimension e Point:

```
public void setSize(Dimension d)
public void setSize(int width, int height)
public void setLocation(Point p)
public void setLocation(int x,int y)
```

Tali proprietà possono essere impostate anche tramite il metodo setBounds(), che accetta come parametri sia una quadrupla di interi sia un oggetto Rectangle:

```
public void setBounds(int x, int y, int width, int height)
public void setBounds(Rectangle r)
```

Ricorrendo al metodo setResizable(boolean b) è possibile stabilire se si vuole permettere all'utente di ridimensionare la finestra manualmente. Infine, vi sono tre metodi piuttosto importanti:

```
public void pack()
public void setVisible(boolean b)
public void setDefaultCloseOperation(int operation)
```

Il primo ridimensiona la finestra tenendo conto delle dimensioni ottimali di ciascuno dei componenti presenti all'interno. Il secondo permette di visualizzare o di nascondere la finestra. Il terzo imposta l'azione da eseguire alla pressione del bottone close, con quattro impostazioni disponibili: WindowConstants.DO_NOTHING_ON_CLOSE (nessun effetto), WindowConstants.HIDE_ON_CLOSE (nasconde la finestra), WindowConstants.DISPOSE_ON_CLOSE (chiude la finestra e libera le risorse di sistema) e JFrame.EXIT_ON_CLOSE (chiude la finestra e conclude l'esecuzione del programma).

Un breve programma è sufficiente a illustrare come si possa: creare un JFrame; assegnargli un titolo, una posizione sullo schermo e una dimensione; stabilirne il comportamento in chiusura; renderlo visibile.

```
import javax.swing.*;

public class JframeExample {
   public static void main(String argv[]) {
      JFrame j = new JFrame();
      j.setTitle("JFrameExample");
      j.setBounds(10, 10, 300, 200);
      j.setDefaultCloseOperation(JFrame. EXIT_ON_CLOSE);
      j.setVisible(true);
   }
}
```

## JDialog

Le finestre di dialogo si usano per consentire l'inserimento di valori, o per segnalare all'utente una situazione anomala. Ogni finestra di dialogo *appartiene* a un'altra finestra: se si chiude il frame principale, anche i JDialog di sua proprietà verranno chiusi. Se si definisce come *modale* un JDialog, alla sua comparsa esso bloccherà il frame di appartenenza, in modo da costringere l'utente a portare a termine l'interazione. Per creare una finestra di dialogo, è necessario specificare tra i parametri del costruttore il reference alla finestra principale, definire il titolo e indicare se si tratta o meno di una finestra modale:

```
JDialog(Dialog owner, String title, boolean modal)
JDialog(Frame owner, String title, boolean modal)
```

Esistono anche costruttori con un numero inferiore di parametri. I metodi presentati per JFrame sono validi anche con JDialog. Naturalmente, non è possibile selezionare l'opzione EXIT_ON_CLOSE con il metodo setDefaultCloseOperation().

# Gerarchia di contenimento

Un'interfaccia grafica è composta da un top level container, tipicamente un JFrame, e da un insieme di componenti disposti al suo interno. Esistono alcuni componenti che hanno il preciso compito di fungere da contenitori per altri componenti. Il più usato di questi è senza dubbio JPanel, un pannello di uso estremamente generale. Con poche righe si può creare un JPanel e inserire un JButton al suo interno:

```
JPanel p = new JPanel();
JButton b = new JButton("Button");
p.add(Component c)
```

Più in generale, è possibile creare un JPanel, disporre alcuni controlli grafici al suo interno grazie al metodo add(Component c) e quindi inserirlo in un altro JPanel o nel content pane di un top level container. Un breve programma permetterà di illustrare meglio i concetti appena esposti e di introdurre i successivi: si tratta di un esempio abbastanza avanzato, e non c'è da preoccuparsi se alcuni aspetti all'inizio dovessero risultare poco chiari:

```
import javax.swing.*;
import java.awt.*;

public class FirstExample {

  public static void main(String argv[]) {
    // Componenti
    JLabel label = new JLabel("Un programma Swing");
    JCheckBox c1 = new JCheckBox("Check Box 1");
    JCheckBox c2 = new JCheckBox("Check Box 2");
    JButton okButton = new JButton("OK");
    JButton cancelButton = new JButton("Cancel");

    // Pannello NORTH
    JPanel northPanel = new JPanel();
    northPanel.add(label);

    // Pannello CENTER
    JPanel centralPanel = new JPanel();
    centralPanel.setLayout(new GridLayout(0,1));
    centralPanel.setBorder(BorderFactory.createEmptyBorder(20, 20, 50, 50));
    centralPanel.add(c1);
    centralPanel.add(c2);

    // Pannello SOUTH
    JPanel southPanel = new JPanel();
    southPanel.setLayout(new FlowLayout(FlowLayout.RIGHT));
```

```
        southPanel.add(cancelButton);
        southPanel.add(okButton);

        // Top Level Container
        JFrame f = new JFrame("First Example");
        f.getContentPane().setLayout(new BorderLayout());
        f.getContentPane().add(BorderLayout.NORTH, northPanel);
        f.getContentPane().add(BorderLayout.CENTER, centralPanel);
        f.getContentPane().add(BorderLayout.SOUTH, southPanel);
        f.pack();
        f.setVisible(true);
    }
}
```

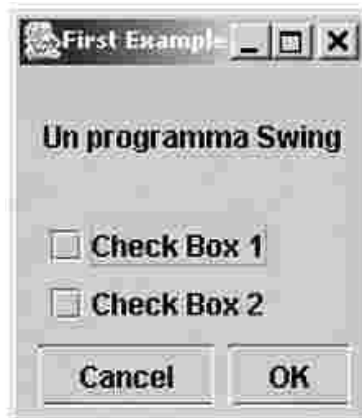
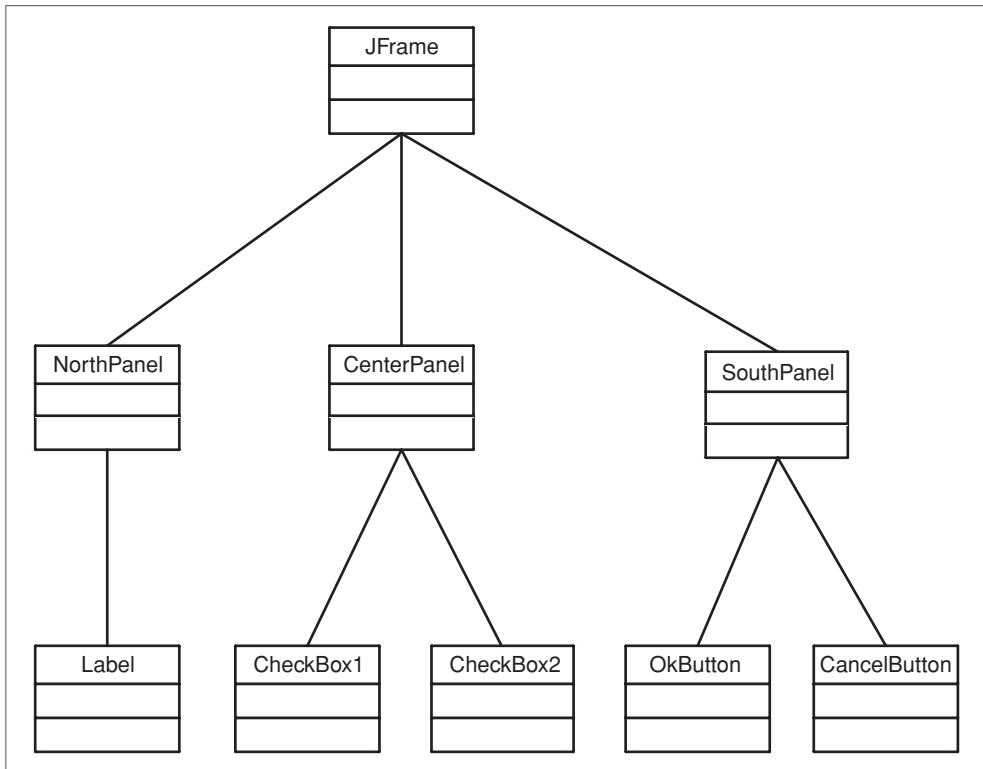
**Figura 12.1 –** *Un programma di esempio.*

Osservando il sorgente si può notare che dapprima vengono creati i componenti, poi sono montati su pannelli e infine tali pannelli vengano disposti all'interno di un JFrame. Per ogni componente vengono impostati alcuni attributi, come il bordo, il testo o il layout manager. È possibile rappresentare con un albero la disposizione gerarchica dei componenti di questo programma, come si vede in figura 12.2.

**Figura 12.2 –** *Gerarchia di contenimento dell'interfaccia grafica del programma di esempio.*

# Layout management

Quando si dispongono i componenti all'interno di un contenitore sorge il problema di come gestire il posizionamento: infatti, sebbene sia possibile specificare le coordinate assolute di ogni elemento dell'interfaccia, queste possono cambiare nel corso della vita del programma allorquando la finestra principale venga ridimensionata.

Per semplificare il lavoro di impaginazione e risolvere questo tipo di problemi  possibile ricorrere ai layout manager, oggetti che si occupano di gestire la strategia di posizionamento dei componenti all'interno di un contenitore.

Il metodo setLayoutManager(LayoutManager m) permette di assegnare un layout manager a ogni pannello. Combinando gli effetti dei layout manager  possibile ottenere la disposizione desiderata.

## FlowLayout

Questo semplice layout manager dispone i componenti in maniera ordinata da sinistra a destra e dall'alto in basso, assegnando a ciascun componente la dimensione minima necessaria a disegnarlo.

```
import java.awt.*;
import javax.swing.*;

public class FlowLayoutExample {
  public static void main(String argv[]) {
    JFrame f = new JFrame();
    f.getContentPane().setLayout(new FlowLayout());
    f.getContentPane().add(new JButton("Primo"));
    f.getContentPane().add(new JButton("Secondo"));
    f.getContentPane().add(new JButton("Terzo"));
    f.getContentPane().add(new JButton("Quarto"));
    f.getContentPane().add(new JButton("Quinto"));
    f.pack();
    f.setVisible(true);
  }
}
```

**Figura 12.3 –** *FlowLayout dispone i componenti da sinistra a destra.*

Al termine di una riga, i componenti vengono inseriti nella successiva. In ogni riga, inoltre, i componenti vengono centrati. Si noti come ogni pulsante assuma una dimensione diversa, corrispondente alla sua dimensione minima.

**Figura 12.4 –** *Ridimensionando il contenitore, i componenti vengono disposti su più righe.*

possibile specificare nel costruttore un criterio di allineamento dei componenti diverso da quello di default: i valori possibili sono FlowLayout.LEFT, FlowLayout.CENTER e FlowLayout.RIGHT.

## GridLayout

GridLayout suddivide il contenitore in una griglia di celle di uguali dimensioni. Le dimensioni della griglia vengono definite mediante il costruttore:

```
public GridLayout(int rows, int columns)
```

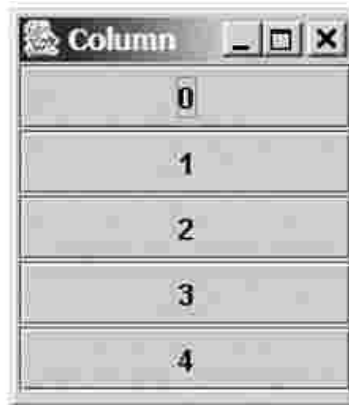
in cui i parametri rows e columns specificano rispettivamente le righe e le colonne della griglia. A differenza di quanto avviene con FlowLayout, i componenti all nterno della griglia assumono automaticamente la stessa dimensione, dividendo equamente lo spazio disponibile. Un semplice esempio permette di illustrare il funzionamento di questo pratico layout manager:

```
import java.awt.*;
import javax.swing.*;

public class GridLayoutExample {
  public static void main(String argv[]) {
    JFrame f = new JFrame("GridLayout");
    f.getContentPane().setLayout(new GridLayout(4, 4));
    for ( int i = 0; i < 14; i++)
      f.getContentPane().add(new JButton(String.valueOf(i)));
    f.pack();
    f.setVisible(true);
  }
}
```

**Figura 12.5 –** *GridLayout.*

È possibile utilizzare GridLayout anche per disporre i componenti in riga o in colonna. Per ottenere questo, è sufficiente impostare i parametri row e column uno a 0 e l'altro a 1:

```
import java.awt.*;
import javax.swing.*;

public class ColumnLayoutExample {
  public static void main(String argv[]) {
    JFrame f = new JFrame("Column");
    f.getContentPane().setLayout(new GridLayout(0, 1));
    for ( int i = 0; i < 5; i++)
      f.getContentPane().add(new JButton(String.valueOf(i)));
    f.pack();
    f.setVisible(true);
  }
}
```

**Figura 12.6 –** *Layout in colonna con GridLayout.*

## BorderLayout

BorderLayout suddivide il contenitore esattamente in cinque aree, disposte a croce. Il programmatore può decidere in quale posizione aggiungere un controllo utilizzando il metodo add(component c,String s), presente nei Container, dove il primo parametro specifica il componente da aggiungere e il secondo indica la posizione. I valori validi per il secondo parametro sono le costanti BorderLayout.NORTH, BorderLayout.SOUTH, BorderLayout.CENTER, BorderLayout.EAST e BorderLayout.WEST. Si osservi un esempio:

```
import java.awt.*;
import javax.swing.*;

public class BorderLayoutExample {
```

```
    public static void main(String argv[]) {
      JFrame f = new JFrame("BorderLayout");
      f.getContentPane().setLayout(new BorderLayout());
      f.getContentPane().add(new Button("North"), BorderLayout.NORTH);
      f.getContentPane().add(new Button("South"), BorderLayout.SOUTH);
      f.getContentPane().add(new Button("East"), BorderLayout.EAST);
      f.getContentPane().add(new Button("West"), BorderLayout.WEST);
      f.getContentPane().add(new Button("Center"), BorderLayout.CENTER);
      f.setSize(500,400);
      f.setVisible(true);
    }
  }
```

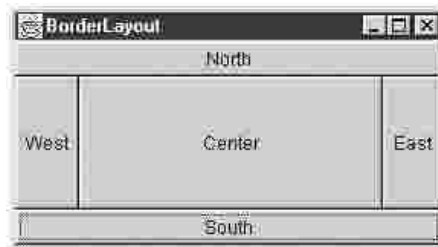
**Figura 12.7 –** *Disposizione standard dei componenti all'interno di un pannello con BorderLayout.*

Si noti che non è obbligatorio riempire tutti e cinque gli spazi: se qualcuno di essi viene lasciato vuoto, i componenti presenti riempiranno lo spazio disponibile.

**Figura 12.8 –** *Comportamento di BorderLayout quando alcuni spazi vengono lasciati vuoti.*

BorderLayout particolarmente indicato per gestire l'impaginazione complessiva di una finestra. Infatti, la sua conformazione ripropone il layout di una tipica applicazione a finestre, con la

barra degli strumenti in alto, una barra di stato in basso, un browser a sinistra o a destra e il pannello principale al centro.

# Progettazione top down di interfacce grafiche

Durante la progettazione delle interfacce grafiche, può essere utile ricorrere a un approccio top down, descrivendo l'insieme dei componenti a partire dal componente più esterno per poi procedere a mano a mano verso quelli più interni. Si può sviluppare una GUI come quella dell'esempio precedente seguendo questa procedura:

1. Si definisce il tipo di top level container su cui si vuole lavorare (tipicamente un JFrame).

2. Si assegna un layout manager al content pane del JFrame, in modo da suddividerne la superficie in aree più piccole.

3. Per ogni area messa a disposizione dal layout manager è possibile definire un nuovo JPanel. Ogni sotto pannello può utilizzare un layout manager differente.

4. Ogni pannello identificato nel terzo passaggio può essere sviluppato ulteriormente, creando al suo interno ulteriori pannelli o disponendo dei controlli.

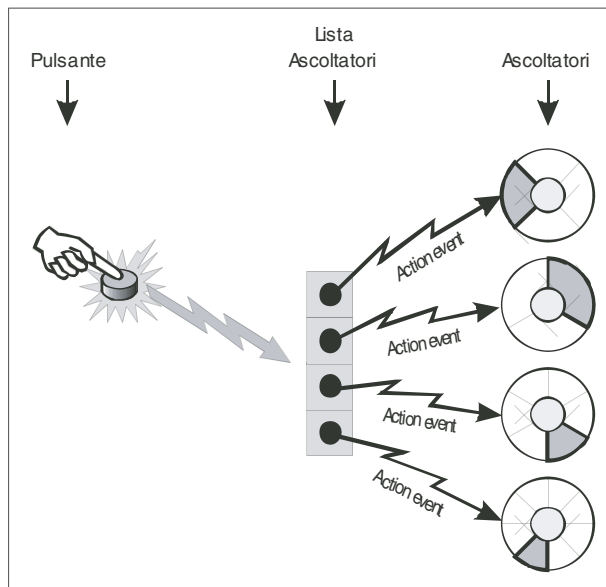
Una volta conclusa la fase progettuale, si può passare a scrivere il codice relativo all'interfaccia: in questo secondo momento, è opportuno adottare un approccio bottom up, realizzando dapprima il codice relativo ai componenti atomici, quindi quello dei contenitori e infine quello del JFrame.

# La gestione degli eventi

La gestione degli eventi grafici in Java segue il paradigma event forwarding (conosciuto anche come event delegation). Ogni oggetto grafico predisposto a essere sollecitato in qualche modo dall'utente genera eventi che vengono inoltrati ad appositi ascoltatori, che reagiscono agli eventi secondo i desideri del programmatore. l'event forwarding presenta il vantaggio di separare la sorgente degli eventi dal comportamento a essi associato: un componente *non sa* (e non è interessato a sapere) cosa avverrà al momento della sua sollecitazione: esso si limita a *notificare* ai propri ascoltatori che l'evento che essi attendevano è avvenuto, e questi provvederanno a produrre l'effetto desiderato.

Ogni componente dispone di un metodo addXxxListener() e removeXxxListener() per ogni evento supportato: per esempio, i pulsanti hanno i metodi addActionListener() e removeActionListener() per gestire gli ascoltatori di tipo ActionListener. Mediante questi metodi è possibile registrare un ascoltatore presso il componente. Nel momento in cui il componente viene sollecitato, esso chiama gli ascoltatori in modalità call back, passando come parametro della chiamata un'apposita sottoclasse di Event che contiene tutte le informazioni significative per l'evento stesso. Ogni ascoltatore è caratterizzato da una particolare interfaccia Java: grazie a questa particolarità,

qualsiasi classe può comportarsi come un ascoltatore, a patto che fornisca un'implementazione per i metodi definiti dalla corrispondente interfaccia listener.

**Figura 12.9 –** *Gestione degli eventi secondo il modello event forwarding.*

Le classi che permettono di gestire gli eventi generati dai componenti Swing sono in gran parte gli stessi utilizzati dai corrispondenti componenti AWT, e si trovano nel package java.awt.event. Alcuni componenti Swing, tuttavia, non hanno un omologo componente AWT: in questo caso le classi necessarie a gestirne gli eventi sono presenti nel package javax.swing.event,

Nel seguente esempio viene creato un pulsante; ad esso viene abbinato un ascoltatore grazie al metodo addActionListener(). Si noti che il gestore dell'evento è la classe EventHandlingExample stessa: essa infatti implementa la classe ActionListener e definisce il metodo actionPerformed(ActionEvent e), al cui interno si trova il codice che verrà richiamato in modalità call back ogni volta che il pulsante viene premuto.

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class EventHandlingExample extends JFrame implements ActionListener {

  public EventHandlingExample() {
    super("Eventi");
```

```
    setSize(400,300);
    JButton b = new JButton("Button");
    getContentPane().add(b);
    b.addActionListener(this);  // aggiunta dell'ascoltatore
  }
  // metodo call back per la gestione dell'evento
  public void actionPerformed(ActionEvent e) {
    JOptionPane.showMessageDialog(this,"pulsante premuto");
  }
  public static void main(String argv[]) {
    EventHandlingExample e = new EventHandlingExample();
    e.setVisible(true);
  }
}
```

Ogni componente grafico può avere più di un ascoltatore per un determinato evento. In questo caso, gli ascoltatori saranno chiamati uno per volta secondo l'ordine in cui si sono registrati. Gli eventi generati dall'utente vengono accodati automaticamente. Se l'utente cerca di scatenare un nuovo evento prima che il precedente sia stato consumato (per esempio premendo ripetutamente un pulsante), ogni nuovo evento verrà accodato, e l'azione corrispondente sarà eseguita solo al termine della precedente.

Il sistema grafico di Java definisce un gran numero di ascoltatori: tra i più importanti vale la pena di segnalare:

- quelli che gestiscono gli eventi legati alle finestre (WindowFocusListener, WindowListener, WindowStateListener);
- quelli relativi ai contenitori (ComponentListener, ContainerListener, FocusListener);
- i listener per gli eventi del mouse (MouseListener, MouseMotionListener, MouseWheelListener);
- e quelli per la tastiera (KeyListener).

Nei prossimi capitoli verranno descritti i principali componenti Java, e per ogni componente verrà illustrato in dettaglio il relativo ascoltatore.

## Uso di adapter nella definizione degli ascoltatori

Alcuni componenti grafici generano eventi così articolati da richiedere, per la loro gestione, ascoltatori caratterizzati da più di un metodo. Per esempio, l'interfaccia MouseMotionListener, che ascolta i movimenti del mouse, ne dichiara due, mentre MouseListener, che ascolta gli eventi relativi ai pulsanti del mouse, ne definisce tre. Infine, l'interfaccia WindowListener ne dichiara addirittura sette, ossia uno per ogni possibile cambiamento di stato della finestra:

```
public interface WindowListener extends EventListener {
   public void windowOpened(WindowEvent e);
   public void windowClosing(WindowEvent e);
   public void windowClosed(WindowEvent e);
   public void windowIconified(WindowEvent e);
   public void windowDeiconified(WindowEvent e);
   public void windowActivated(WindowEvent e);
   public void windowDeactivated(WindowEvent e);
}
```

Il linguaggio Java obbliga a fornire un'implementazione per ciascun metodo definito da un'interfaccia; se si desidera creare un ascoltatore interessato a un solo evento (per esempio uno che intervenga quando la finestra viene chiusa), esso dovrà comunque fornire un'implementazione vuota anche dei metodi cui non è interessato. Nei casi come questo è possibile ricorrere agli adapter: classi di libreria che forniscono un'implementazione vuota di un determinato ascoltatore. Per esempio, il package java.awt.event contiene la classe WindowAdapter, che fornisce un'implementazione vuota di tutti i metodi previsti dall'interfaccia. Una sottoclasse di WindowAdapter, pertanto, è un valido WindowListener, con in più il vantaggio di una maggior concisione:

```
class WindowClosedListener extends WindowAdapter {
  public void windowClosed(WindowEvent e) {
     // codice da eseguire alla chiusura della finestra
  }
}
```

All'interno dei package java.awt.event e javax.swing.event sono presenti adapter per ogni ascoltatore dotato di più di un metodo:

```
ComponentAdapter
ContainerAdapter
FocusAdapter
HierarchyBoundsAdapter
InternalFrameAdapter
KeyAdapter
MouseAdapter
MouseInputAdapter
MouseMotionAdapter
WindowAdapter
```

## Classi anonime per definire gli ascoltatori

Le classi anonime forniscono un metodo molto sintetico per creare ascoltatori, utile nelle situazioni in cui sia necessario crearne parecchie decine. Le classi anonime sono classi prive di nome che vengono definite nello stesso momento in cui vengono create. Ecco un esempio di poche righe, che mostra come creare un pulsante e aggiungervi un ActionListener:

```
JButton b = new JButton("Button");
b.addActionListener(new ActionListener() {
  public void actionPerformed(ActionEvent e) {
    JOptionPane.showMessageDialog(this,"pulsante premuto");
  }
});
```

Si osservi come all'interno del metodo addActionListener() venga creata un'istanza di una classe di tipo ActionListener, il cui codice è contenuto tra le parentesi graffe che seguono. Una classe di questo tipo viene detta anonima proprio a causa del fatto che non ne viene definito il nome, ma solamente il corpo. Le classi anonime sono state introdotte nel linguaggio Java a partire dal JDK 1.1, proprio a supporto del sistema di event forwarding: esse, infatti, consentono di gestire in modo rapido numerose situazioni molto comuni nella programmazione a eventi.

Capitolo 13

# Bottoni e menu

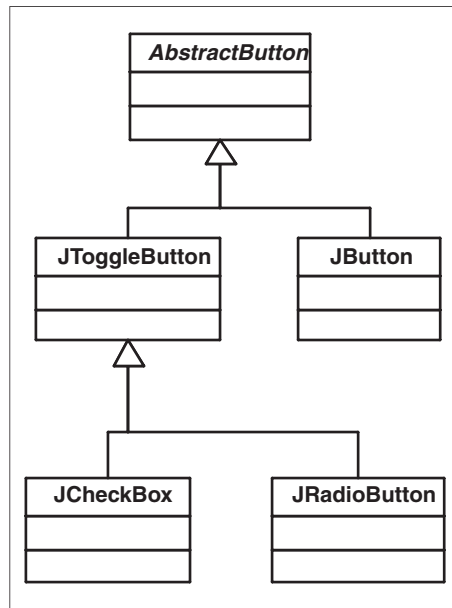
ANDREA GINI

## Pulsanti

I pulsanti, grazie alla loro modalità di utilizzo estremamente intuitiva, sono sicuramente i controlli grafici più usati. Il package Swing offre quattro tipi di pulsanti, legati tra loro dalla gerarchia illustrata in figura 13.2.

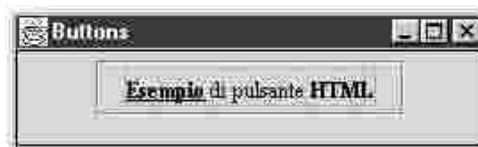
**Figura 13.1 –** *Pulsanti disponibili in Java.*

JButton è l'implementazione del comune bottone push; JToggleButton è un pulsante on/off; JCheckBox è una casella di controllo, ossia un controllo grafico creato sul modello delle caselle di spunta dei questionari; JRadioButton serve a creare pulsanti di opzione, che permettono di scegliere una possibilità tra molte in modo mutuamente esclusivo.

**Figura 13.2 –** *Gerarchia dei principali pulsanti Java.*

## AbstractButton: gestione dell'aspetto

La classe AbstractButton definisce l'interfaccia di programmazione comune a tutti i pulsanti. L'API di AbstractButton definisce un insieme di metodi per gestire l'aspetto del componente. In particolare, viene fornita la possibilità di associare a ogni controllo grafico un'etichetta di testo, un'icona o entrambi. È possibile impostare l'etichetta in formato HTML: basta aggiungere il prefisso <html> nel parametro di setText(String). Le seguenti righe di codice mostrano come creare un JButton con un'etichetta HTML:

```
JButton b = new JButton();
b.setText("<html><font size=-1><b><u>Esempio</u></b> di pulsante <b>HTML</b></font></html>"));
```

**Figura 13.3 –** *Su tutti i pulsanti Swing è possibile impostare un'etichetta HTML.*

I pulsanti Swing permettono di impostare un'icona diversa per ognuno degli stati in cui si possono trovare: normale, premuto, selezionato (valido per i controlli che mantengono lo stato come i CheckBox), disabilitato e rollover (lo stato in cui si trova il pulsante quando viene sorvolato dal puntatore del mouse).

```
void setIcon(Icon defaultIcon)
void setPressedIcon(Icon pressedIcon)
void setSelectedIcon(Icon selectedIcon)
void setDisabledIcon(Icon disabledIcon)
void setRolloverIcon(Icon rolloverIcon)
void setDisabledSelectedIcon(Icon disabledSelectedIcon)
void setRolloverSelectedIcon(Icon rolloverSelectedIcon)
```

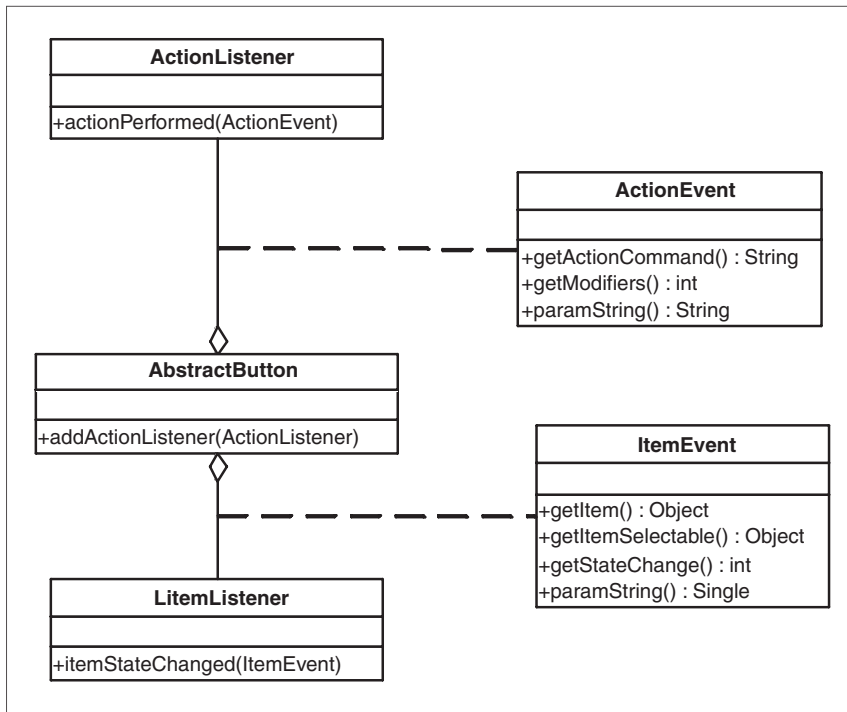
Come icona si può utilizzare un oggetto di tipo ImageIcon. Per farlo, si ricorre al costruttore ImageIcon(String filename) che crea un'icona a partire da un'immagine di tipo GIF o JPEG, il cui percorso viene specificato nella stringa del parametro. Se non viene specificato diversamente, le immagini per gli stati premuto, selezionato e disabilitato vengono create in modo automatico a partire da quella di default. Le seguenti righe creano un pulsante decorato con l'immagine img.gif (può andare bene una qualsiasi immagine GIF o JPEG). Naturalmente, il file deve essere presente nella directory di lavoro, altrimenti verrà creato un pulsante vuoto:

```
JButton b = new JButton();
b.setIcon(new ImageIcon("img.gif"));
```

Altri metodi importanti sono void setText(String text), che permette di impostare la scritta sopra il pulsante, setEnabled(boolean b) che permette di abilitare o disabilitare il componente, e setRolloverEnabled(boolean b) che attiva o disattiva l'effetto rollover.

## Eventi dei pulsanti

I controlli derivati da AbstractButton prevedono due tipi di ascoltatore: ActionListener e ItemListener. Il primo ascolta l'evento realtivo alla pressione del pulsante. Il secondo ascolta invece i cambiamenti tra gli stati selezionato e non selezionato, ed è utile nei pulsanti di tipo JToggleButton. Nei paragrafi seguenti verranno illustrati esempi di uso di entrambi gli ascoltatori.

**Figura 13.4 –** *Pulsanti Java, ascoltatori ed eventi.*

# JButton

JButton, il comune pulsante push, è la più importante sottoclasse di AbstractButton. I seguenti costruttori permettono di creare pulsanti e di definirne le principali proprietà visuali, ossia l'etichetta di testo e l'icona:

```
JButton(String text)
JButton(Icon icon)
JButton(String text, Icon icon)
```

Ogni top level container può segnalare un pulsante come DefaultButton. Esso verrà evidenziato in modo particolare e richiamato con la semplice pressione del tasto Invio. Le righe seguenti creano un pulsante, un JFrame e impostano il pulsante come DefaultButton.

```
JFrame f = new JFrame();
JButton b = new JButton("DefaultButton");
f.getContentPane().add(b);ty
f.getRootPane().setDefaultButton(b);
```

L'ascoltatore più utile per un JButton è ActionListener, che riceve eventi di tipo ActionEvent quando il pulsante viene premuto. Si vedrà ora un programma di esempio che illustra l'uso di due JButton, uno dei quali viene registrato come DefaultButton, e dei relativi ascoltatori.

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class JButtonExample extends JFrame {

  private JButton dialogButton;
  private JButton closeButton;

  public JButtonExample() {
    super("JButtonExample");
    closeButton = new JButton("Close");
    dialogButton = new JButton("Open Frame");
    dialogButton.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        JOptionPane.showMessageDialog(closeButton, "Chiudi questa finestra per proseguire");
      }
    });
    closeButton.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        try {
          System.exit(0);
        }
        catch (Exception ex) {
        }
      }
    });
    getContentPane().setLayout(new FlowLayout(FlowLayout.CENTER));
    getRootPane().setDefaultButton(closeButton);
    getContentPane().add(closeButton);
    getContentPane().add(dialogButton);
    pack();
  }

  public static void main(String argv[]) {
    JButtonExample x = new JButtonExample();
    x.setVisible(true);
  }
}
```

**Figura 13.5 –** *Il programma JButtonExample.*

# JToggleButton

JToggleButton è un pulsante che può trovarsi in due stati: premuto e rilasciato. Cliccando con il mouse si provoca il passaggio tra uno stato e l'altro. I costruttori permettono di creare JToggleButton e di impostarne le proprietà:

```
JToggleButton(String text, Icon icon, boolean selected)
JToggleButton(String text, boolean selected)
JToggleButton(Icon icon, boolean selected)
```

Il parametro selected permette di impostare lo stato iniziale del pulsante. Un JToggleButton, quando viene premuto, genera un ActionEvent e un ItemEvent. L'evento più significativo per questo tipo di pulsanti è il secondo, che segnala il cambiamento di stato, ossia il passaggio da premuto a rilasciato e viceversa. La classe ItemEvent dispone di due metodi importanti: Object getItem() e int getStateChange(). Il primo restituisce un reference al pulsante che ha generato l'evento (è necessario ricorrere al casting); il secondo sostituisce un intero che può assumere i valori ItemEvent.SELECTED oppure ItemEvent.DESELECTED, a seconda del valore assunto dal componente. È comunque possibile utilizzare ascoltatori di tipo ActionListener in modo simile all'esempio precedente. L'esempio che segue crea una finestra con un JToggleButton, che permette di aprire e di chiudere una finestra di dialogo non modale:

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class JToggleButtonExample extends JFrame {

  private JDialog dialog;
  private JToggleButton jDialogButton;

  public JToggleButtonExample() {
```

```
    super("JToggleButtonExample");
    setBounds(10, 35, 250, 70);
    FlowLayout fl = new FlowLayout(FlowLayout.CENTER);
    getContentPane().setLayout(fl);
    dialog = createDialog();

    jDialogButton = new JToggleButton("Open / Close Frame", false);
    jDialogButton.addItemListener(new JDialogButtonItemListener());
    getContentPane().add(jDialogButton);
    setVisible(true);
  }
  public JDialog createDialog() {
    JDialog d = new JDialog(this, "JDialog", false);
    d.setBounds(250, 20, 300, 100);
    d.getContentPane().setLayout(new BorderLayout());
    d.getContentPane().add(BorderLayout.CENTER, new JLabel("Finestra Aperta", JLabel.CENTER));
    d.setDefaultCloseOperation(JFrame.DO_NOTHING_ON_CLOSE);
    return d;
  }
  // Ascoltatore di JDialogButton
  class JDialogButtonItemListener implements ItemListener {
    public void itemStateChanged(ItemEvent e) {
      int status = e.getStateChange();
      if ( status == ItemEvent.SELECTED )
        dialog.setVisible(true);
      else
        dialog.setVisible(false);
    }
  }

  public static void main(String argv[]) {
    JToggleButtonExample b = new JToggleButtonExample();
  }
}
```

**Figura 13.6 –** *Il programma JToggleButtonExample.*

Il codice dell'ascoltatore JDialogButtonItemListener è un po' più complesso di quello degli ActionListener dell'esempio precedente. Questo tipo di ascoltatore deve normalmente prevedere una verifica dello stato in cui si trova il pulsante, al fine di produrre la reazione appropriata. La verifica dello stato viene effettuata interrogando l'oggetto ItemEvent con il metodo getStateChange().

## JCheckBox

JCheckBox è una sottoclasse di JToggleButton che crea caselle di controllo, con un aspetto simile a quello delle caselle di spunta dei questionari. Il suo funzionamento è analogo a quello della superclasse, ma di fatto tende a essere utilizzato in contesti in cui si offre all'utente la possibilità di scegliere una o più opzioni tra un insieme, come avviene per esempio nei pannelli di controllo. I costruttori disponibili sono gli stessi di JToggleButton e così pure la gestione degli eventi, quindi non sarà necessario ripetere quanto è stato già detto. Un esempio mostrerà un uso tipico di questo componente:

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class JCheckBoxExample extends JFrame {

  private JDialog dialog1;
  private JDialog dialog2;
  private JCheckBox checkBox1;
  private JCheckBox checkBox2;

  public JCheckBoxExample() {
    // Imposta le proprietà del top level container
    super("JCheckBoxExample");
    setBounds(10, 35, 200, 70);
    getContentPane().setLayout(new FlowLayout(FlowLayout.CENTER));

    // Crea due finestre di dialogo non modali,
    // inizialmente invisibili
    dialog1 = new JDialog(this, "JDialog 1", false);
    dialog1.setBounds(250, 20, 300, 100);
    dialog1.getContentPane().setLayout(new BorderLayout());
    dialog1.getContentPane().add(BorderLayout.CENTER new JLabel("Finestra 1 Aperta", JLabel.CENTER));
    dialog1.setDefaultCloseOperation(JFrame.DO_NOTHING_ON_CLOSE);
    dialog2 = new JDialog(this, "JDialog 2", false);
    dialog2.setBounds(250, 150, 300, 100);
    dialog2.getContentPane().setLayout(new BorderLayout());
    dialog2.getContentPane().add(BorderLayout.CENTER, new JLabel("Finestra 2 Aperta", JLabel.CENTER));
    dialog2.setDefaultCloseOperation(JFrame.DO_NOTHING_ON_CLOSE);

    // Crea i checkBox e li registra presso il loro ascoltatore
```

```
        ItemListener listener = new JCheckBoxItemListener();
        checkBox1 = new JCheckBox("Finestra 1");
        checkBox1.addItemListener(listener);
        checkBox2 = new JCheckBox("Finestra 2");
        checkBox2.addItemListener(listener);

        // Aggiunge i checkBox al top level container
        getContentPane().add(checkBox1);
        getContentPane().add(checkBox2);
        setVisible(true);
    }
    // ascoltatore JCheckBox
    class JCheckBoxItemListener implements ItemListener {
        public void itemStateChanged(ItemEvent e) {
            Object target = e.getItem();
            int status = e.getStateChange();

            if(target.equals(checkBox1) && status == ItemEvent.SELECTED)
                        dialog1.setVisible(true);
            else if(target.equals(checkBox1) && status == ItemEvent.DESELECTED)
                        dialog1.setVisible(false);
            else if(target.equals(checkBox2) && status == ItemEvent.SELECTED)
                        dialog2.setVisible(true);
            else if(target.equals(checkBox2) && status == ItemEvent.DESELECTED)
                        dialog2.setVisible(false);
        }
    }
    public static void main(String argv[]) {
        JCheckBoxExample b = new JCheckBoxExample();
    }
}
```

**Figura 13.7 –** *Il programma JCheckBoxExample.*

L'ascoltatore JCheckBoxItemListener presenta un grado di complessità maggiore del precedente. Esso infatti ascolta entrambi i controlli, e a ogni chiamata verifica quale dei due abbia generato l'evento, chiamando il metodo getItem() di ItemEvent, e quale stato esso abbia assunto, predisponendo la reazione opportuna.

# JRadioButton

JRadioButton è una sottoclasse di JToggleButton, dotata dei medesimi costruttori. Questo tipo di controllo, chiamato pusante di opzione, viene usato tipicamente per fornire all'utente la possibilità di operare una scelta tra un insieme di possibilità, in contesti nei quali un'opzione esclude l'altra.

Per implementare questo comportamento di mutua esclusione, è necessario registrare i JRadioButton che costituiscono l'insieme presso un'istanza della classe ButtonGroup, come viene mostrato nelle righe seguenti:

```
ButtonGroup group = new ButtonGroup();
group.add(radioButton1);
group.add(radioButton2);
group.add(radioButton3);
```

Ogni volta che l'utente attiva uno dei pulsanti registrati presso il ButtonGroup, gli altri vengono automaticamente messi a riposo. Questo comportamento ha una conseguenza importante nella gestione degli eventi. Infatti, un gruppo di JRadioButton registrati presso un ButtonGroup genera *due* ItemEvent consecutivi per ogni clic del mouse: uno per la casella che viene selezionata e uno per quella deselezionata. Di norma, si è interessati unicamente al fatto che un particolare JRadioButton sia stato premuto, poiché la politica di mutua esclusione rende superflua la verifica dello stato. In questi casi, è consigliabile utilizzare un ActionListener come nell'esempio seguente, nel quale un gruppo di JRadioButton permette di modificare la scritta su un'etichetta di testo:

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class JRadiobuttonExample extends JFrame {

  private JRadioButton radioButton1;
  private JRadioButton radioButton2;
  private JRadioButton radioButton3;
  private JLabel label;

  public JRadiobuttonExample() {
    // Imposta le proprietà del top level container
    super("JRadiobuttonExample");
```

```
        setBounds(10, 35, 150, 150);
        getContentPane().setLayout(new FlowLayout(FlowLayout.CENTER));

        // Crea i radiobutton e la label
        radioButton1 = new JRadioButton("RadioButton 1");
        radioButton2 = new JRadioButton("RadioButton 2");
        radioButton3 = new JRadioButton("RadioButton 3");
        label = new JLabel();

        // Crea l'ascoltatore e registra i JRadioButton
        ActionListener listener = new JRadioButtonListener();
        radioButton1.addActionListener(listener);
        radioButton2.addActionListener(listener);
        radioButton3.addActionListener(listener);

        // Crea il ButtonGroup e registra i RadioButton
        ButtonGroup group = new ButtonGroup();
        group.add(radioButton1);
        group.add(radioButton2);
        group.add(radioButton3);

        // Aggiunge i componenti al top level container
        getContentPane().add(radioButton1);
        getContentPane().add(radioButton2);
        getContentPane().add(radioButton3);
        getContentPane().add(label);

        radioButton1.doClick();
        setVisible(true);
    }
    // Ascoltatore JRadioButton
    class JRadioButtonListener implements ActionListener {
        public void actionPerformed(ActionEvent e) {
            String target = e.getActionCommand();
            label.setText(target);
        }
    }
    public static void main(String argv[]) {
        JRadiobuttonExample b = new JRadiobuttonExample();
    }
}
```

**Figura 13.8 –** *Il programma JRadioButtonExample.*

## JToolBar

Nelle moderne interfacce grafiche, l'insieme dei controlli viene suddiviso tra due luoghi: la Menu Bar, di cui si parlerà più avanti, e la Tool Bar, di cui ci si occupa ora. JToolBar è un contenitore che permette di raggruppare un insieme di controlli grafici in una riga, che solitamente viene posizionata al di sotto della barra dei menu. Sebbene sia utilizzata soprattutto come contenitore di pulsanti provvisti di icona, è possibile inserire al suo interno qualsiasi tipo di componente, come campi di testo o elenchi di selezione a discesa.

Ricorrendo al drag & drop è possibile staccare una Tool Bar dalla sua posizione originale e renderla fluttuante: in questo caso, essa verrà visualizzata in una piccola finestra separata dal frame principale. Allo stesso modo, è possibile afferrare una barra degli strumenti con il mouse e trascinarla in una nuova posizione.

L'uso di JToolBar all'interno dei propri programmi non presenta particolari difficoltà. È sufficiente crearne un'istanza, aggiungervi i componenti nell'ordine da sinistra a destra e posizionarla all'interno del contenitore principale:

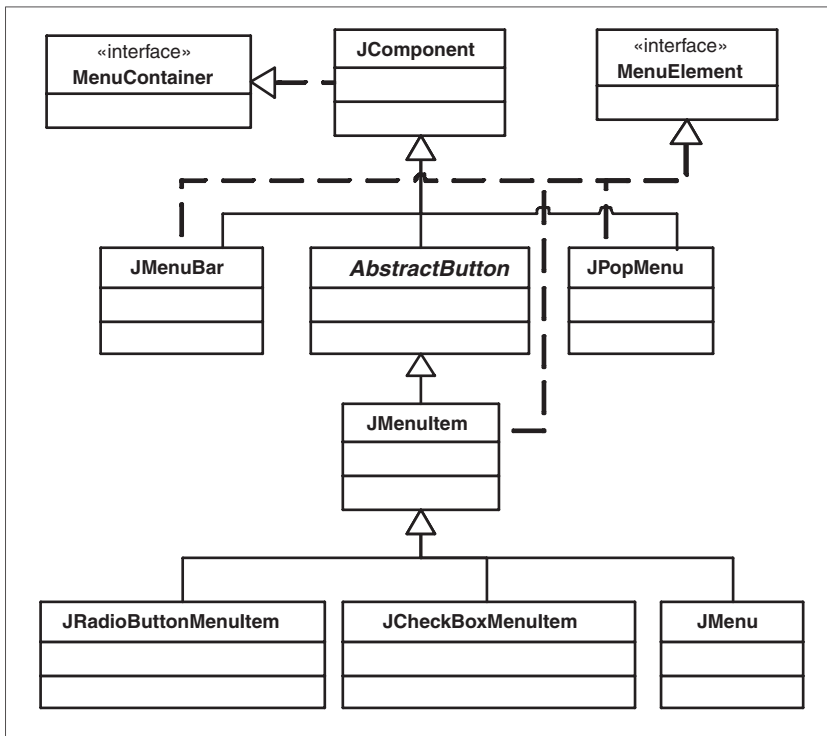
```
JToolBar toolBar = new JToolBar();
JButton b = new JButton(new ImageIcon("img.gif"));
toolBar.add(b);
....
JFrame f = new JFrame();
f.getContentPane().setLayout(new BorderLayout());
f.getContentPane().add(BorderLayout.NORTH, toolBar);
```

Tra i metodi di JToolBar, vale la pena segnalare add(Component c), che permette di aggiungere un componente alla barra, e addSeparator(), che aggiunge un segnale di separazione.

# I menu

I menu sono controlli che permettono di accedere a un grande numero di opzioni in uno spazio ridotto, organizzato gerarchicamente. Ogni programma grafico dispone di una Menu Bar

organizzata per gruppi di funzioni: accesso al disco, operazioni sulla clipboard, opzioni e così via. Ogni menu può contenere sia elementi terminali (MenuItem) sia ulteriori menu nidificati.

**Figura 13.9 –** *Gerarchia dei componenti di tipo menu.*

In Swing anche i menu si assemblano in modo gerarchico, costruendo un oggetto per ogni elemento e aggiungendolo al proprio contenitore. La gerarchia delle classi in figura 13.8 mostra che ogni sottoclasse di JComponent è predisposta a contenere menu, capacità che viene garantita dall'interfaccia MenuContainer. Le classi JMenu e JPopupMenu sono contenitori appositamente realizzati per questo scopo. La classe JMenuItem implementa l'elemento di tipo più semplice: essendo sottoclasse di AbstractButton, ne eredita l'interfaccia di programmazione e il comportamento (vale a dire che tutti i metodi visti nel paragrafo AbstractButton possono essere utilizzati anche su questi componenti). JRadioButtonMenuItem e JCheckBoxMenuItem sono analoghi ai pulsanti JRadioButton e JCheckBox; oltre alla parentela diretta con JMenuItem, essi hanno in comune con JMenu e JPopupMenu l'interfaccia MenuElement, che accomuna tutti i componenti che possono comparire all'interno di un menu. Il metodo add(JMenu m), comune a tutte le classi, permette di innestare qualsiasi menu all'interno di qualunque altro. I seguenti costruttori permettono di

creare JMenuItem, JRadioButtonMenuItem e JCheckBoxMenuItem in maniera simile a come si può fare con i JButton. I parametri permettono di specificare l'etichetta, l'icona e lo stato:

```
JMenuItem(String text)
JMenuItem(String text, Icon icon)
JCheckBoxMenuItem(String text, Icon icon, boolean b)
JCheckBoxMenuItem(String text, boolean b)
JRadioButtonMenuItem(String text, boolean selected)
JRadioButtonMenuItem(String text, Icon icon, boolean selected)
```

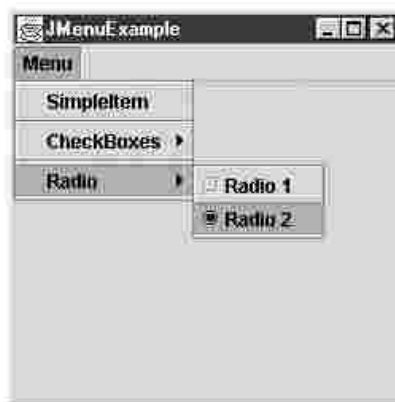
Sebbene sia possibile posizionare una JMenuBar ovunque all'interno di un'interfaccia grafica, i top level container JFrame, JApplet e JDialog riservano a questo scopo una posizione esclusiva, situata appena sotto la barra del titolo. È possibile aggiungere un JMenu a un JFrame, a un JApplet o a un JDialog mediante il metodo setJMenuBar(JMenuBar), come si vede nelle righe di esempio:

```
JMenuBar menubar = new JMenuBar();
....
JFrame f = new JFrame("A Frame");
f.setJMenuBar(menuBar)
```

La gestione degli eventi nei menu è del tutto simile a quella dei pulsanti: ogni volta che si seleziona un JMenuItem, esso lancia un ActionEvent ai suoi ascoltatori. Normalmente si usa ActionListener per i JMenuItem e ItemListener per i JCheckboxMenuItem, mentre per JRadioButton-MenuItem è possibile usare sia l'uno sia l'altro. Il seguente esempio illustra la costruzione di un menu ricorrendo a elementi di ogni tipo:

```
import javax.swing.*;

public class JMenuExample extends JFrame {
  public JMenuExample() {
    // Imposta le proprietà del top level container
    super("JMenuExample");
    setBounds(10, 35, 250, 250);
    // Crea menu, sottomenu e menuitems
    JMenuBar menubar = new JMenuBar();
    JMenu menu = new JMenu("Menu");
    JMenuItem simpleItem = new JMenuItem("SimpleItem");
    JMenu checkSubMenu = new JMenu("CheckBoxes");
    JCheckBoxMenuItem check1 = new JCheckBoxMenuItem("Check 1");
    JCheckBoxMenuItem check2 = new JCheckBoxMenuItem("Check 1");
    JMenu radioSubMenu = new JMenu("Radio");
    JRadioButtonMenuItem radio1 = new JRadioButtonMenuItem("Radio 1");
    JRadioButtonMenuItem radio2 = new JRadioButtonMenuItem("Radio 2");
    ButtonGroup group = new ButtonGroup();
```

```
      group.add(radio1);
      group.add(radio2);
      // Compone i menu
      checkSubMenu.add(check1);
      checkSubMenu.add(check2);
      radioSubMenu.add(radio1);
      radioSubMenu.add(radio2);
      menu.add(simpleItem);
      menu.addSeparator();// (new JSeparator());
      menu.add(checkSubMenu);
      menu.addSeparator();//.add(new JSeparator());
      menu.add(radioSubMenu);
      menubar.add(menu);
      // Aggiunge la barra dei menu al JFrame
      setJMenuBar(menubar);
      setVisible(true);
    }
    public static void main(String argv[]) {
      JMenuExample m = new JMenuExample();
    }
  }
```

**Figura 13.10 –** *Il programma JMenuExample.*

## JPopupMenu

I JPopupMenu implementano i menu contestuali presenti in quasi tutti i moderni sistemi a finestre. La costruzione di JPopupMenu è del tutto simile a quella di Jmenu, mentre diversa è la modalità di visualizzazione. Il metodo:

```
public void show(Component invoker, int x, int y)
```

visualizza il menu al di sopra del componente specificato dal parametro invoker, alle coordinate x e y (relative a invoker). Per associare un JPopupMenu alla pressione del pulsante destro del mouse su un oggetto grafico, è necessario registrare il componente interessato presso un MouseListener incaricato di chiamare il metodo show() al momento opportuno. Dal momento che alcuni sistemi a finestre mostrano il menu contestuale alla pressione del pulsante destro (evento mousePressed), mentre altri lo mostrano al momento del rilascio (evento mouseReleased), è bene ascoltare entrambi gli eventi, controllando la condizione isPopupTrigger() sull'evento MouseEvent. Esso infatti restituisce true solamente se l'evento corrente è quello che provoca il richiamo del menu contestuale nella piattaforma ospite:

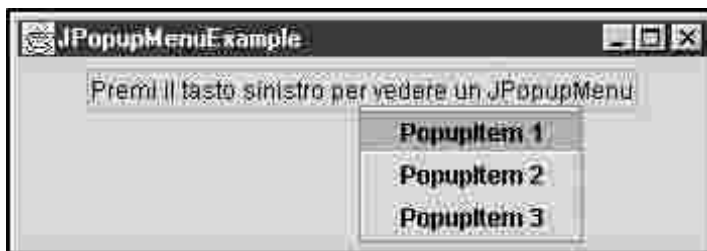
```
class PopupListener extends MouseAdapter {
  // pulsante destro premuto (stile Motif)
  public void mousePressed(MouseEvent e) {
    if (e.isPopupTrigger()) {
        popup.show(e.getComponent(), e.getX(), e.getY());
    }
  }
  // pulsante destro premuto e rilasciato (stile Windows)
  public void mouseReleased(MouseEvent e) {
    if (e.isPopupTrigger()) {
      popup.show(e.getComponent(), e.getX(), e.getY());
    }
  }
}
```

Questo accorgimento permette di creare programmi che rispecchiano il comportamento della piattaforma ospite, senza ambiguità che potrebbero disorientare l'utente.

Il seguente esempio crea un JTextField, cui aggiunge un MouseListener che si occupa di visualizzare un JPopupMenu alla pressione del pulsante destro:

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class JPopupMenuExample extends JFrame {
  private JPopupMenu popup;

  public JPopupMenuExample() {
    super("JPopupMenuExample");
    setBounds(10, 35, 350, 120);

    JTextField textField = new JTextField("Premi il pulsante sinistro per vedere un JPopupMenu");
    textField.setEditable(false);
```

```
        getContentPane().setLayout(new FlowLayout());
        getContentPane().add(textField);

        popup = new JPopupMenu();
        JMenuItem popupItem1 = new JMenuItem("PopupItem 1");
        JMenuItem popupItem2 = new JMenuItem("PopupItem 2");
        JMenuItem popupItem3 = new JMenuItem("PopupItem 3");
        popup.add(popupItem1);
        popup.add(popupItem2);
        popup.add(popupItem3);

        // Aggiunge un MouseListener al componente
        // che deve mostrare il menu
        MouseListener popupListener = new PopupListener();
        textField.addMouseListener(popupListener);
        setVisible(true);
    }
    class PopupListener extends MouseAdapter {
        public void mousePressed(MouseEvent e) {
            if (e.isPopupTrigger()) {
                popup.show(e.getComponent(), e.getX(), e.getY());
            }
        }
        public void mouseReleased(MouseEvent e) {
            if (e.isPopupTrigger()) {
                popup.show(e.getComponent(), e.getX(), e.getY());
            }
        }
    }
    public static void main(String[] args) {
        JPopupMenuExample window = new JPopupMenuExample();
    }
}
```

**Figura 13.11 –** *Il programma JPopupMenuExample.*

# Gestire gli eventi con le action

La maggior parte dei programmi grafici permette di accedere a una funzionalità in diverse maniere. I word processor, per esempio, consentono di effettuare un *cut* su clipboard in almeno tre modi distinti: dal menu Modifica, tramite il pulsante identificato dall'icona della forbice o tramite una voce del menu contestuale. Questa ridondanza è gradita all'utente, che ha la possibilità di utilizzare il programma secondo le proprie abitudini e il proprio grado di esperienza, ma può rivelarsi complicata da implementare per il programmatore. In Swing è possibile risolvere questo genere di problemi ricorrendo alle Action: oggetti che permettono di associare un particolare evento a un gruppo di controlli grafici, fornendo nel contempo la possibilità di gestire in modo centralizzato gli attributi e lo stato.

## Descrizione dell'API

L'interfaccia Action, sottoclasse di ActionListener, eredita il metodo actionPerformed(ActionEvent e), con il quale si implementa la normale gestione degli eventi. Il metodo setEnabled(boolean b) permette di abilitare una Action; la chiamata a questo metodo provoca automaticamente l'aggiornamento dello stato di tutti i controlli grafici a esso associati. La coppia di metodi:

```
Object getValue(String key)
void putValue(String key, Object value)
```

permette di leggere o impostare coppie chiave-valore (in modo simile a quanto avviene nelle Hashtable) in cui la chiave è una stringa che descrive un attributo e il valore è l'attributo stesso. Tra le possibili chiavi si possono segnalare le seguenti:

```
Action.NAME
Action.SHORT_DESCRIPTION
Action.SMALL_ICON
```

Esse permettono di specificare, rispettivamente, il nome dell'azione (che verrà riportato sul pulsante), il testo da usare nei ToolTip e l'icona da esporre sui controlli abbinati alla Action. Per esempio, se si desidera impostare l'icona relativa a una Action, occorre utilizzare l'istruzione putValue in questo modo:

```
action.putValue(Action.SMALL_ICON, new ImageIcon("img.gif"))
```

La classe AbstractAction, implementazione dell'interfaccia Action, fornisce costruttori che permettono di impostare le proprietà in modo più intuitivo:

```
AbstractAction(String name)
AbstractAction(String name, Icon icon)
```

# Uso delle Action

È possibile creare un oggetto Action estendendo la classe AbstractAction e fornendo il codice del metodo actionPerformed(ActionEvent e), in modo simile a quanto si farebbe per un oggetto di tipo ActionListener:

```
class MyAction extends AbstractAction {
  private Icon myIcon = new ImageIcon("img.gif");
  public MyAction() {
    super("My Action", myIcon);
  }
  public void actionPerformed(ActionEvent e) {
    // qui va il codice dell'ascoltatore
  }
}
```

È possibile definire una Action come classe anonima:

```
Action myAction = new AbstractAction("My Action", new ImageIcon("img.gif")) {
  public void actionPerformed(ActionEvent e) {
    // qui va il codice dell'ascoltatore
  }
});
```

Per abbinare una Action ai corrispondenti controlli grafici è sufficiente utilizzare il metodo add(Action a), presente in JMenu, JToolBar e JPopupMenu, come si vede nelle seguenti righe:

```
Action myAction = new MyAction();
JToolBar toolBar = new JToolBar();
JMenuBar menuBar = new JMenuBar();
JPopupMenu popup = new JPopupMenu();
// aggiunge un pulsante alla Tool Bar
toolBar.add(myAction);
// aggiunge un MenuItem alla Menu Bar
menuBar.add(myAction);
// aggiunge un MenuItem al Popup Menu
popup.add(myAction);
```

Dal momento che il metodo add(Action) restituisce il componente che viene creato, è possibile cambiarne l'aspetto anche dopo la creazione. Se si vuole aggiungere un MenuItem al menu, ma si desidera che esso sia rappresentato soltanto da una stringa di testo, senza icona, è possibile agire nel modo seguende:

```
JMenuItem mi = menuBar.add(myAction);
mi.setIcon(null);
```

Se durante l'esecuzione del programma si vogliono disabilitare i controlli abbinati a MyAction, è possibile farlo ricorrendo all'unica istruzione:

```
myAction.setEnabled(false);
```

che provvederà a disabilitare tutti i controlli legati a MyAction.

# Controlli per inserimento dati

Andrea Gini

## Tipologie di controlli

### JTextField

I JTextField (campi di testo) sono oggetti grafici che permettono di editare una singola riga di testo. Premendo il tasto Invio viene generato un ActionEvent, per segnalare agli ascoltatori che il testo è stato immesso. È possibile creare un JTextField mediante i seguenti costruttori:

```
JTextField()
JTextField(String text)
```

I seguenti metodi, invece, permettono di impostare o di leggere le principali proprietà dell'oggetto:

```
setText(String text)
setColumns(int columns)
setFont(Font f)
```

L'ascoltatore di default per JTextField è ActionListener, che viene invocato alla pressione del tasto Invio. Gli oggetti ActionEvent generati da un JTextField permettono di accedere direttamente al testo contenuto nel componente tramite il metodo getActionCommand(). L'esempio seguente mostra come creare un JTextField e un ascoltatore che reagisca all'inserimento di testo:

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class JTextFieldExample extends JFrame {
   private JTextField textField;
   private JLabel label;

   public JTextFieldExample() {
      super("JTextField");
      setSize(200, 80);
      getContentPane().setLayout(new BorderLayout());

      textField = new JTextField();
      label = new JLabel();
      getContentPane().add(BorderLayout.NORTH, textField);
      textField.addActionListener(new EnterTextListener());
      getContentPane().add(BorderLayout.SOUTH, label);

      setVisible(true);
   }
   class EnterTextListener implements ActionListener {
      public void actionPerformed(ActionEvent e) {
         label.setText("Testo inserito: " + textField.getText());
         textField.setText("");
      }
   }
   public static void main(String argv[]) {
      JTextFieldExample jtf = new JTextFieldExample();
   }
}
```

**Figura 14.1 –** *Il programma JTextFieldExample.*

## JPasswordField

JPasswordField è una sottoclasse di JTextField specializzata nell'inserimento di password. Le principali differenze rispetto alla superclasse sono due: la prima è che in JPasswordField i caratteri digitati vengono visualizzati di default tramite asterischi (***); la seconda è che il testo in chiaro viene

restituito sotto forma di array di char e non come stringa. Il metodo setEchoChar(char c) permette di impostare qualsiasi carattere al posto dell'asterisco di default. Il metodo char[] getPassword(), invece, restituisce il contenuto del campo di testo in chiaro, sotto forma di array di char.

**Figura 14.2 –** *JPasswordField impedisce di vedere quello che l'utente sta digitando.*

## JComboBox

I JComboBox offrono all'utente la possibilità di effettuare una scelta a partire da un elenco di elementi, anche molto lungo. A riposo il componente si presenta come un pulsante, con l'etichetta corrispondente al valore attualmente selezionato. Un clic del mouse provoca la comparsa di un menu provvisto di barra laterale di scorrimento, che mostra le opzioni disponibili. Se si imposta un JComboBox come editabile, esso si comporterà a riposo come un JTextField, permettendo all'utente di inserire valori non presenti nella lista.

È possibile creare un JComboBox usando i seguenti costruttori. Il secondo costruttore permette di inizializzare il componente con una lista di elementi di tipo String, Icon o JLabel.

```
JComboBox()
JComboBox(Object[] items)
```

Un gruppo di metodi permette di aggiungere, togliere o manipolare gli elementi dell'elenco, così come si fa con un Vector:

```
void addItem(Object anObject
void removeItem(Object anObject)
void removeItemAt(int anIndex)
void removeAllItems()
Object getItemAt(int index)
int getItemCount()
void insertItemAt(Object anObject, int index)
```

Per operare sull'elemento correntemente selezionato, sono disponibili tre metodi che permettono di ottenere l'elemento (che viene restituito sotto forma di Object) e di impostarlo mediante un indice numerico o l'oggetto stesso:

```
Object getSelectedItem()
void setSelectedIndex(int anIndex)
void setSelectedItem(Object anObject)
```

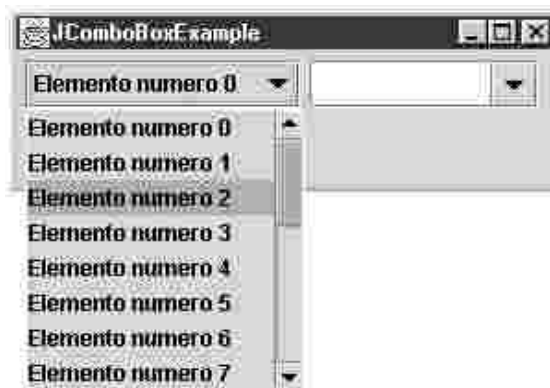
L'ultimo metodo interessante è quello che permette di rendere un JComboBox editabile:

```
void setEditable(boolean b)
```

Nel seguente esempio vengono creati due JComboBox: uno editabile e l'altro no. All'interno del primo è possibile inserire gli elementi digitandoli direttamente nel componente e premendo Invio. Come ascoltatori vengono usati due ActionListener: EditListener si occupa di aggiungere alla lista i nuovi elementi, mentre SelectionListener viene invocato da entrambi i componenti al fine di aggiornare una JLabel con il valore dell'elemento selezionato.

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class JComboBoxExample extends JFrame {
  private JComboBox uneditableComboBox;
  private JLabel label;
  private JComboBox editableComboBox;
  private String[] items;

  public JComboBoxExample() {
    // Imposta le proprietà del top level container
    super("JComboBoxExample");
    setBounds(10, 35, 300, 100);
    getContentPane().setLayout(new FlowLayout(FlowLayout.LEFT));

    // Crea 20 elementi
    items = new String[20];
    for(int i = 0; i < 20; i++)
      items[i] = "Elemento numero " + String.valueOf(i);

    // Inizializza un ComboBox non editabile
    uneditableComboBox = new JComboBox(items);
    ActionListener selectionListener = new SelectionListener();
    uneditableComboBox.addActionListener(selectionListener);

    label = new JLabel();

    // Inizializza un JComboBox editabile
    editableComboBox = new JComboBox();
    editableComboBox.setEditable(true);
    editableComboBox.addActionListener(new EditListener());
    editableComboBox.addActionListener(selectionListener);

    getContentPane().add(uneditableComboBox);
    getContentPane().add(editableComboBox);
    getContentPane().add(label);
```

```
        setVisible(true);
    }

    class SelectionListener implements ActionListener {
        public void actionPerformed(ActionEvent e) {
            JComboBox cb = (JComboBox)e.getSource();
            String selectedItem = (String)cb.getSelectedItem();
            label.setText("Selezionato: " + selectedItem);
        }
    }
    class EditListener implements ActionListener {
        public void actionPerformed(ActionEvent e) {
            JComboBox cb = (JComboBox)e.getSource();
            String selectedItem = (String)cb.getSelectedItem();
            editableComboBox.addItem(selectedItem);
            editableComboBox.setSelectedItem("");
        }
    }
    public static void main(String argv[]) {
        JComboBoxExample b = new JComboBoxExample();
    }
}
```

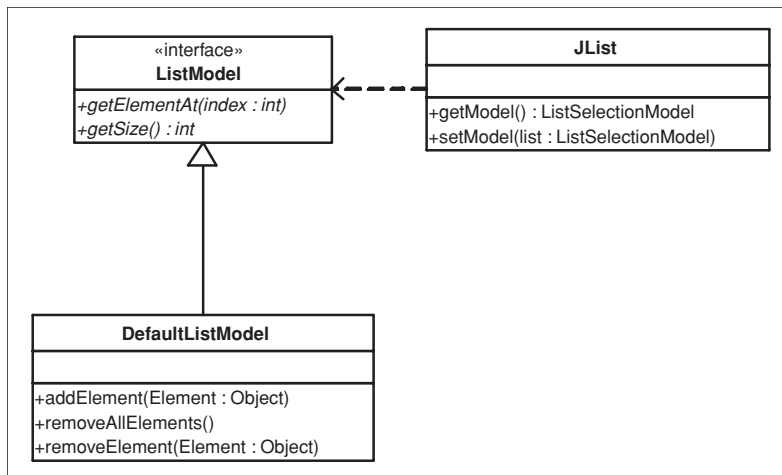
**Figura 14.3 –** *JComboBox permette di scegliere un elemento da un elenco.*

# JList

JList è un altro componente che permette di scegliere tra elementi che compongono un elenco. Diversamente da JComboBox consente di selezionare più di un elemento per volta, utilizzando il tasto Shift per selezionare elementi contigui o il tasto Ctrl per elementi separati.

Per utilizzare in modo completo JList è necessario comprendere la sua struttura interna: come

si può vedere dalla figura 14.4, JList mantiene gli elementi dell'elenco in un oggetto conforme all'interfaccia ListModel. Il package javax.swing contiene DefaultListModel, un'implementazione di JList di uso generico che permette di aggiungere o togliere a piacere elementi dall'elenco. Contrariamente a quanto sembra suggerire il nome, DefaultListModel *non* è il modello di default: se si crea una JList a partire da un vettore, esso utilizzerà un proprio ListModel non modificabile, al quale non potranno essere aggiunti o tolti elementi. Se si vuole creare una JList più flessibile, occorre procedere nel modo seguente:

```
listModel = new DefaultListModel();
listModel.addElement("Elemento 1");
listModel.addElement("Elemento 2");
....
list = new JList(listModel);
```

**Figura 14.4 –** *Diagramma delle classi di JList.*

Per visualizzare correttamente JList è indispensabile montarlo all'interno di un JScrollPane (un pannello dotato di barra di scorrimento) e aggiungere quest'ultimo al pannello principale. In caso contrario, non sarà possibile visualizzare tutti gli elementi presenti nell'elenco.

```
list = new JList(listModel);
JScrollPane scroll = new JScrollPane(list);
panel.add(scroll);
```

I costruttori permettono di creare un JList a partire da un oggetto di tipo ListModel o da un generico vettore di oggetti:

```
JList(ListModel dataModel)
JList(Object[] listData)
```

Come già accennato, il secondo costruttore produrrà una JList non modificabile. Gli elementi del vettore listData possono essere di tipo String, JLabel o Icon. Qualsiasi altro oggetto verrà visualizzato tramite la stringa restituita dal metodo toString().

Il seguente metodo permette di impostare la modalità di selezione:

```
void setSelectionMode(int selectionMode)
```

Il parametro selectionMode può assumere i seguenti valori: JList.SINGLE_SELECTION se si desidera che sia possibile selezionare un solo elemento per volta; JList.SINGLE_INTERVAL_SELECTION se si vuole permettere la selezione di un singolo intervallo per volta; JList.MULTIPLE_INTERVAL_SELECTION se non si vuole porre restrizioni al numero di elementi o intervalli selezionabili.

Un gruppo di metodi permette di operare sull'elemento selezionato, quando questo è unico:

```
Object getSelectedValue()
int getSelectedIndex()
void setSelectedIndex(int index)
boolean isSelectedIndex(int index)
void clearSelection()
```

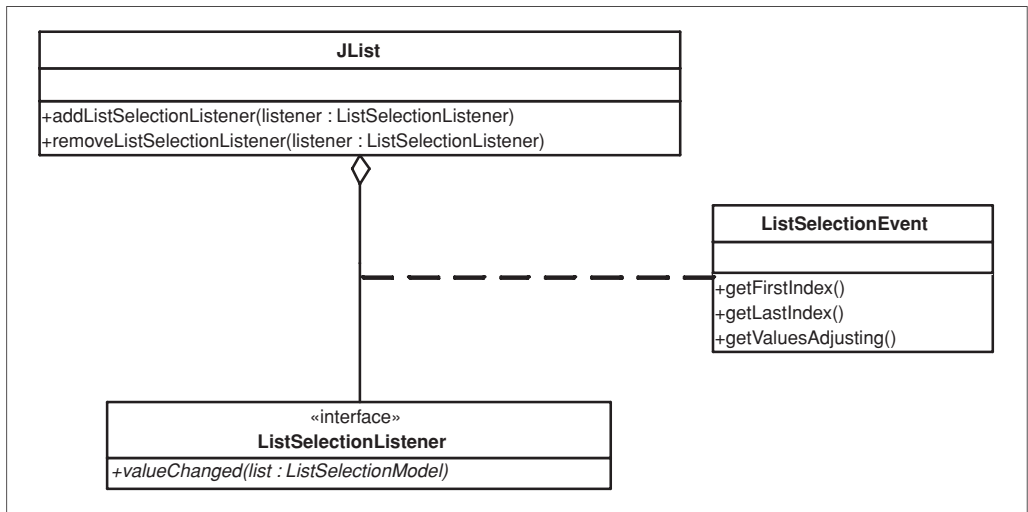
Il metodo getSelectedIndex() restituisce l'indice del primo elemento selezionato, o il valore -1 se al momento non è selezionato alcun elemento. isSelectedIndex(int index) consente invece di sapere se un determinato elemento è selezionato o meno. Una quadrupla di metodi permette di operare su selezioni multiple:

```
int[] getSelectedIndices()
Object[] getSelectedValues()
void setSelectedIndices(int[] indices)
void setSelectionInterval(int anchor, int lead)
```

Ogni volta che l'utente seleziona un elemento, viene notificato un ListSelectionEvent ai ListSelectionListener registrati. Per gestire l'elenco degli ascoltatori sono disponibili i caratteristici metodi:

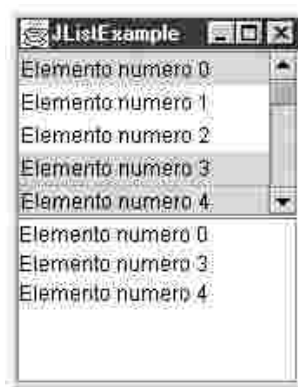
```
void addListSelectionListener(ListSelectionListener listener)
void removeListSelectionListener(ListSelectionListener listener)
```

I metodi di ListSelectionEvent permettono di conoscere gli indici di inizio e di fine della selezione. Tramite il metodo boolean getValueIsAdjusting() è possibile sapere se l'utente sta ancora operando sulla selezione, o se ha terminato rilasciando il pulsante del mouse.

**Figura 14.5 –** *Modello di eventi di JList.*

L'esempio seguente crea un JList con 20 elementi selezionabili a intervalli non contigui, e un'area di testo non modificabile che elenca gli elementi attualmente selezionati:

```
import javax.swing.*;
import javax.swing.event.*;
import java.awt.*;

public class JListExample extends JFrame {

  private JList list;
  private JTextArea output;

  public JListExample() {
    super("JListExample");
    setSize(170, 220);
    getContentPane().setLayout(new GridLayout(0, 1));

    // Crea 20 elementi
    String[] items = new String[20];
    for(int i = 0; i < 20;i++)
      items[i] = "Elemento numero " + String.valueOf(i);

    // Inizializza una JList
    list = new JList(items);
    list. setSelectionMode(ListSelectionModel.MULTIPLE_INTERVAL_SELECTION);
```

```
      ListSelectionListener selectionListener = new SelectionListener();
      list.addListSelectionListener(selectionListener);

      // Crea la TextArea di output
      output = new JTextArea();
      output.setEditable(false);

      // assembla la GUI
      getContentPane().add(new JScrollPane(list));
      getContentPane().add(new JScrollPane(output));
      setVisible(true);
   }
   class SelectionListener implements ListSelectionListener {
      public void valueChanged(ListSelectionEvent e) {
         if(!e.getValueIsAdjusting()) {
            JList list = (JList)e.getSource();
            output.setText(“”);
            Object[] selectedItems = list.getSelectedValues();
            for(int i = 0; i < selectedItems.length; i++)
                     output.append(selectedItems[i].toString() + “\n”);
            }
         }
      }
   public static void main(String argv[]) {

      JListExample b = new JListExample();
   }
}
```

**Figura 14.6 –** *Premendo il tasto Ctrl è possibile selezionare più di un elemento dall'elenco.*

# JSlider

JSlider è un cursore a slitta, che permette di inserire in maniera continua valori numerici compresi tra un massimo e un minimo, eliminando di fatto la possibilità di inserire valori scorretti. Le proprietà più importanti di un JSlider sono il valore minimo, il valore massimo e l'orientamento, che può essere orizzontale o verticale. Il costruttore principale permette di specificare questi attributi al momento della creazione, e di impostare la posizione iniziale del cursore:

```
JSlider(int orientation, int min, int max, int value)
```

Le quattro proprietà fondamentali del componente possono essere impostate anche mediante i seguenti metodi:

```
void setOrientation(int orientation)
void setMinimum(int min)
void setMaximum(int max)
void setValue(int value)
```

Alcuni metodi permettono di impostare altre proprietà visuali del componente:

```
void setInverted(boolean b)
void setPaintTicks(boolean b)
void setPaintLabels(boolean b)
```

Il primo di questi metodi permette di scegliere se la scala debba essere disegnata da destra a sinistra (true) o da sinistra verso destra (false); il secondo se disegnare o meno il righello; il terzo, infine, se disegnare le etichette. Una coppia di metodi consente di impostare la spaziatura tra le tacche del righello:

```
void setMajorTickSpacing(int)
void setMinorTickSpacing(int)
```

Esiste infine la possibilità di personalizzare ulteriormente l'aspetto del componente, specificando etichette non regolari. Per farlo, bisogna chiamare il metodo setLabelTable(HashTable h), passando come parametro una HashTable che contenga coppie chiave-valore composte da un oggetto di tipo Int e un Component: ogni Component verrà disegnato in corrispondenza della tacca specificata dell'intero passato come chiave. Nelle righe seguenti viene mostrata la creazione di un JSlider con etichette testuali. Naturalmente, è possibile utilizzare al posto delle label qualsiasi tipo di componente, per esempio dei JLabel contenenti icone, o addirittura pulsanti programmati per riposizionare il cursore su valori preimpostati.

```
slider = new JSlider(JSlider.VERTICAL, 0, 70, 15);
slider.setMajorTickSpacing(10);
slider.setPaintTicks(true);
```

```
Hashtable labelTable = new Hashtable();
labelTable.put(new Integer(0), new JLabel("Silence"));
labelTable.put(new Integer(10), new JLabel("Low"));
labelTable.put(new Integer(30), new JLabel("Normal"));
labelTable.put(new Integer(70), new JLabel("Loud!"));
slider.setLabelTable(labelTable);
```

Il metodo Hashtable createStandardLabels(int incremen,int start) permette di creare tabelle con configurazioni standard, a partire dal valore specificato dal parametro start con la progressione data dal parametro increment:

```
s.setLabelTable(s.createStandardLabels(10,5));
```

**Figura 14.7 –** *Con il metodo setLabelTable() è possibile personalizzare il righello.*

## Eventi JSlider

JSlider utilizza ChangeListener come ascoltatore e ChangeEvent come evento. I metodi per la gestione dell'elenco degli ascoltatori sono conformi alle consuete convenzioni di naming:

```
void addChangeListener(ChangeListener l)
void removeChangeListener(ChangeListener l)
```

Un ascoltatore di tipo ChangeListener deve implementare il metodo stateChanged(ChangeEvent e). Per conoscere lo stato di un JSlider bisogna interrogare il componente attraverso due metodi: getValue(), che restituisce il valore intero su cui il cursore è attualmente posizionato; getValueIsAdjusting(), che restituisce true se l'azione di modifica è tuttora in corso. Un ascoltatore come il seguente effettua un'azione solamente quando il cursore viene rilasciato, e scarta tutti gli eventi di aggiustamento:

```
class SliderChangeListener implements ChangeListener {
  public void stateChanged(ChangeEvent e) {
    JSlider slider = (JSlider)e.getSource();
    if(!slider.getValueIsAdjusting())
      label.setText("Selezionato: " + String.valueOf(slider.getValue()));
  }
}
```

## Esempio d'uso

Le seguenti righe di codice chiariranno meglio l'uso di questo componente:

```
slider = new JSlider(JSlider.HORIZONTAL, 0, 60, 15);
slider.setMajorTickSpacing(10);
slider.setMinorTickSpacing(5);
slider.setPaintTicks(true);
slider.setPaintLabels(true);
```

Il costruttore crea un JSlider orizzontale, la cui scala varia tra 0 e 60 con il cursore posizionato sul 15. Seguono due metodi che impostano la spaziatura tra le tacche del righello: il primo imposta la spaziatura tra le tacche più grandi, il secondo tra quelle più piccole. Gli ultimi due metodi attivano il disegno del righello e della guida numerata, normalmente disattivati. Questa sequenza di metodi permette di creare un gran numero di cursori a slitta, come si può vedere nel prossimo esempio:

```
import javax.swing.*;
import java.awt.*;
import javax.swing.event.*;

public class JSliderExample extends JFrame {

  private JSlider slider1;
  private JSlider slider2;
  private JSlider slider3;
  private JLabel label;

  public JSliderExample() {
    super("JSlider");
    setSize(220, 240);
    getContentPane().setLayout(new FlowLayout(FlowLayout.LEFT));
    ChangeListener listener = new SliderChangeListener();
    slider1 = new JSlider(JSlider.HORIZONTAL, 0, 60, 15);
    slider1.setMajorTickSpacing(10);
    slider1.setMinorTickSpacing(5);
    slider1.setPaintTicks(true);
    slider1.setPaintLabels(true);
    slider1.addChangeListener(listener);
    slider2 = new JSlider(JSlider.HORIZONTAL, 0, 60, 10);
    slider2.setMajorTickSpacing(15);
```

```
    slider2.setMinorTickSpacing(5);
    slider2.setPaintTicks(true);
    slider2.setPaintLabels(true);
    slider2.addChangeListener(listener);
    slider3 = new JSlider(JSlider.HORIZONTAL, 0, 60, 30);
    slider3.setMajorTickSpacing(5);
    slider3.setMinorTickSpacing(1);
    slider3.setPaintTicks(true);
    slider3.setPaintLabels(true);
    slider3.addChangeListener(listener);
    label = new JLabel("Selezionato: " + String.valueOf(slider1.getValue()));
    getContentPane().add(slider1);
    getContentPane().add(slider2);
    getContentPane().add(slider3);
    getContentPane().add(label);
    setVisible(true);
  }

  class SliderChangeListener implements ChangeListener {
    public void stateChanged(ChangeEvent e) {
      JSlider source = (JSlider)e.getSource();
      label.setText("Selezionato: " + String.valueOf(source.getValue()));
    }
  }

  public static void main(String argv[]) {
    JSliderExample b = new JSliderExample();
  }
}
```

**Figura 14.8 –** *Il programma JSliderExample.*

# JTextArea

Il package Swing dispone di un insieme completo di componenti di testo. Dopo JTextField, già esaminato all'inizio di questo capitolo, si prenderà ora in esame JTextArea, che crea un'area di testo: un oggetto grafico da utilizzare quando si intende lavorare su testi di lunghezza arbitraria privi di attributi di stile.

La costruzione di una JTextArea non presenta particolarità degne di nota: è importante invece fare attenzione, quando si assembla l'interfaccia grafica, a collocare il componente all'interno di un JScrollPane. In caso contrario, non sarà possibile navigare un testo più lungo di una schermata:

```
JTextArea ta = new JTextArea();
getContentPane().add(BorderLayout.CENTER, new JScrollPane(ta));
```

Una JTextArea può essere editabile o meno: tale comportamento può essere modificato tramite il metodo setEditable(boolean b). Un gruppo di metodi permette di impostare il font e i colori di primo piano e di sfondo:

```
void setFont(Font font)
void setForeground(Color fg)
void setBackground(Color bg)
```

## Manipolazione del testo

È possibile editare il testo nel componente direttamente con la tastiera e il mouse, oppure da programma, ricorrendo ad alcuni metodi:

```
void setText(String t)
void append(String str)
void insert(String str, int pos)
void replaceRange(String str, int start, int end)
```

Il primo di questi permette di impostare il testo del componente, cancellando il contenuto precedente. Il secondo consente invece di aggiungere una stringa in coda al testo preesistente. I metodi insert() e replaceRange(), infine, servono a inserire del testo in una determinata posizione, o a rimpiazzare un frammento di testo, specificato dai parametri start e end, con una stringa. Questi metodi richiedono come parametri valori interi, che fanno riferimento alla posizione del cursore rispetto all'inizio del documento: se la text area in questione contenesse nella prima riga la frase "La vispa Teresa" e nella seconda "avea tra l'erbetta", sarebbe possibile dire che la parola "Teresa" è compresa tra gli offset 9 e 15, mentre "erbetta" si trova tra gli indici 27 e 34 (va contato anche il ritorno carrello).

JTextArea dispone anche di una coppia di metodi che permettono di comunicare con i dispositivi mediante gli stream, al fine di caricare o salvare il testo:

```
void read(Reader in, Object descriptor)
void write(Writer out)
```

Il metodo read() richiede come parametro un descriptor, ossia un oggetto che fornisce informazioni aggiuntive sul formato del testo: dal momento che JTextArea opera unicamente su testo privo di attributi, è possibile impostare tale parametro a null.

Per leggere il contenuto dell'area di testo sono disponibili due versioni del metodo getText(). Una è priva di argomenti e restituisce tutto il testo presente all'interno del componente; l'altra, invece, richiede di specificare la posizione di inizio e la lunghezza del frammento:

```
String getText()
String getText(int offs, int len)
```

## Cursore e selezione

Un gruppo di metodi permette di conoscere la posizione attuale del cursore ed eventualmente di modificarla:

```
int getCaretPosition()
void setCaretPosition(int position)
void moveCaretPosition(int pos)
```

È anche possibile operare selezioni (mettere in evidenza una porzione di testo) in maniera programmatica, indicandone l'inizio e la fine:

```
int getSelectionStart()
int getSelectionEnd()

void select(int selectionStart, int selectionEnd)
void selectAll()
String getSelectedText()
void replaceSelection(String content)
```

Mediante le selezioni è anche possibile operare sulla clipboard di sistema, ricorrendo ai metodi cut(), copy() e paste().

# Pannelli, accessori e decorazioni

Andrea Gini

Si è visto come sia possibile creare interfacce grafiche innestando pannelli uno all'interno dell'altro. La scelta di pannelli disponibili in Swing non è limitata al semplice JPanel, ma include pannelli specializzati nel trattamento di casi particolari. In questo capitolo verranno illustrati JSplitPane e JTabbedPane, due pannelli estremamente utili per creare interfacce grafiche. In un secondo tempo verrà spiegato l'uso della classe JOptionPane, che permette di creare finestre di dialogo di diverso genere, e sarà presentato il meccanismo di gestione dei look & feel di Swing. Per finire, si vedrà un esempio di applicazione grafica di una certa complessità.

## Pannelli

### JSplitPane

JSplitPane è un pannello formato da due aree, separate da una barra mobile. Al suo interno è possibile disporre una coppia di componenti, affiancati lateralmente o uno sopra l'altro. Il divisore può essere trascinato per impostare l'area da assegnare a ciascun componente, rispettandone la dimensione minima. Usando JSplitPane in abbinamento a JScrollPane si può ottenere una coppia di pannelli ridimensionabili. Il seguente programma crea una finestra con un JSplitPane al suo interno: nel pannello superiore monta un'immagine JPEG, in quello inferiore un'area di testo in cui si possono annotare commenti. Per avviarlo è necessario specificare sulla riga di comando il percorso di un file JPEG o GIF (per esempio, java JSplitDemo c:\immagine.jpg):

```
import javax.swing.*;
import java.awt.*;

public class JSplitDemo extends JFrame {

  public JSplitDemo(String fileName) {
     super("JSplitPane");
     setSize(300, 250);

     // costruisce un pannello contenente un'immagine
     ImageIcon img = new ImageIcon(fileName);
     JLabel picture = new JLabel(img);
     JScrollPane pictureScrollPane = new JScrollPane(picture);

     // crea un pannello che contiene un'area di testo
     JTextArea comment = new JTextArea();
     JScrollPane commentScrollPane = new JScrollPane(comment);

     // Crea uno SplitPane verticale con i due pannelli al suo interno
     JSplitPane splitPane= new JSplitPane(JSplitPane.VERTICAL_SPLIT, pictureScrollPane, commentScrollPane);
     splitPane.setOneTouchExpandable(true);
     splitPane.setDividerLocation(190);
     splitPane.setContinuousLayout(true);

     // aggiunge lo SplitPane al frame principale
     getContentPane().add(splitPane);
     setVisible(true);
  }
  public static void main(String argv[]) {

     if(argv.length == 1) {
        JSplitDemo b = new JSplitDemo(argv[0]);
     }
     else
        System.out.println("usage JSplitDemo <filename>");
  }
}
```

**Figura 15.1 –** *Un esempio di JSplitPane.*

## JSplitPane API

Nell'esempio si è fatto ricorso a un costruttore che permette di impostare le più importanti proprietà dello SplitPane con un'unica istruzione:

```
public JSplitPane(int orientamento, Component leftComponent, Component rightComponent)
```

Il primo parametro serve a specificare l'orientamento: esso può assumere i valori JSplitPane.HORIZONTAL_SPLIT o JSplitPane.VERTICAL_SPLIT. Il secondo e il terzo parametro permettono di impostare i due componenti da disporre nel pannello. Questi parametri possono essere impostati anche con i seguenti metodi:

```
void setOrientation(int orientation)
void setBottomComponent(Component comp)
void setTopComponent(Component comp)
void setRightComponent(Component comp)
void setLeftComponent(Component comp)
```

Un gruppo di tre metodi permette di specificare la posizione del divisore:

```
void setDividerLocation(int location)
void setDividerLocation(double proportionalLocation)
void setResizeWeight(double d)
```

Il primo di questi metodi chiede di specificare con un intero la posizione assoluta del divisore, mentre il secondo imposta la posizione del divisore suddividendo in modo proporzionale lo spazio disponibile tra i due componenti (con un valore di 0,5 il divisore viene posto a metà). Infine, il metodo `setResizeWeight(double d)` consente di specificare come si desidera distribuire lo spazio che si viene a creare quando il componente viene ridimensionato, mediante un parametro reale compreso tra 0 e 1. Se si imposta un valore di 0,5 lo spazio in più viene diviso in misura uguale tra i due componenti.

Per modificare l'aspetto fisico del pannello sono disponibili le seguenti possibilità:

```
void setDividerSize(int)
void setOneTouchExpandable(boolean)
void setContinuousLayout(boolean)
```

Il primo serve a impostare la dimensione in pixel della barra di divisione; il secondo permette di attivare una coppia di pulsanti a freccia che consentono di espandere o collassare il divisore con un semplice clic; il terzo, infine, consente di specificare se si desidera che il pannello venga ridisegnato durante il posizionamento del divisore.

Per suddividere un'area in più di due aree ridimensionabili, è possibile inserire i `JSplitPane` uno dentro l'altro. Nel seguente esempio vengono creati un `JSplitPane` orizzontale, contenente una `JTextArea`, e un `JSplitPane` verticale, che contiene a sua volta due ulteriori `JTextArea`.

```
JScrollPane scroll1 = new JScrollPane(new JTextArea());
JScrollPane scroll2 = new JScrollPane(new JTextArea());
JScrollPane scroll3 = new JScrollPane(new JTextArea());

JSplitPane internalSplit
= new JSplitPane(JSplitPane.VERTICAL_SPLIT, scroll1, scroll2);
JSplitPane externalSplit
= new JSplitPane(JSplitPane.HORIZONTAL_SPLIT, scroll3, internalSplit );
```

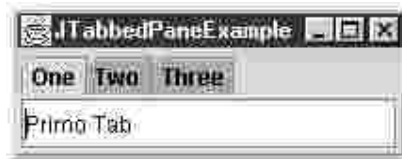
**Figura 15.2 –** *È possibile creare JSplitPane multipli e annidarli l'uno dentro l'altro.*

## JTabbedPane

JTabbedPane permette a diversi componenti di condividere lo stesso spazio sullo schermo: l'utente può scegliere su quale componente operare premendo il Tab corrispondente. Il tipico uso di questo componente è nei pannelli di controllo, nei quali si assegna a ogni Tab la gestione di un insieme di funzioni differenti. Oltre all'innegabile utilità, questo componente presenta una modalità di impiego straordinariamente semplice: è sufficiente creare il pannello e aggiungervi i vari componenti usando il metodo addTab(String title, Component c), in cui il primo parametro specifica l'etichetta del Tab, e il secondo passa il componente. Il passaggio da un Tab all'altro viene ottenuto cliccando con il mouse sul Tab desiderato, senza bisogno di gestire gli eventi in modo esplicito. Nell'esempio seguente viene creato un JTabbedPane al quale vengono aggiunti tre Tab, ognuno dei quali contiene un componente grafico diverso. L'esempio mostra anche un esempio di gestione degli eventi: al cambio di Tab viene aggiornato il titolo della finestra.

```
import javax.swing.*;
import javax.swing.event.*;

public class JTabbedPaneExample extends JFrame {
  private JTabbedPane tabbedPane;
  public JTabbedPaneExample() {
    super("JTabbedPaneExample");
    tabbedPane = new JTabbedPane();

    JTextField tf = new JTextField("primo Tab");
    JButton b = new JButton("secondo Tab");
    JSlider slider = new JSlider(JSlider.HORIZONTAL, 0, 60, 15);
    tabbedPane.addChangeListener(new TabListener());
    tabbedPane.addTab("uno", tf );
    tabbedPane.addTab("due", b );
    tabbedPane.addTab("tre", slider);

    getContentPane().add(tabbedPane);
```

```
        pack();
        setVisible(true);
    }
    public class TabListener implements ChangeListener {
        public void stateChanged(ChangeEvent e) {
            int pos = tabbedPane.getSelectedIndex();
            String title = tabbedPane.getTitleAt(pos);
            setTitle(title);
        }
    }
    public static void main(String[] args) {
     JTabbedPaneExample te = new JTabbedPaneExample();
    }
}
```

**Figura 15.3 –** *Un semplice esempio di JTabbedPane.*

Nell'esempio è stato creato un JTabbedPane e sono stati inseriti al suo interno tre Tab, ognuno dei quali contiene un componente. Naturalmente, è possibile inserire all'interno di un Tab un intero pannello con tutto il suo contenuto:

```
JPanel panel = new JPanel();
panel.setLayout(new BorderLayout());
panel.add(BorderLayout.NORTH, new Button("nord"));
....
panel.add(BorderLayout.SOUTH, new Button("sud"));
tabbedPane.addTab("Pannello", panel);
....
```

## JTabbedPane API

Per creare un JTabbedPane è possibile ricorrere ai costruttori riportati di seguito:

```
JTabbedPane()
JTabbedPane(int tabPlacement)
```

Il secondo permette di specificare il posizionamento dei tab attraverso un parametro che può assumere i valori JTabbedPane.TOP, JTabbedPane.BOTTOM, JTabbedPane.LEFT o JTabbedPane.RIGHT.

Per aggiungere o togliere componenti è disponibile un gruppo di metodi simili a quelli di una nomale collection, tipo Vector. Ogni pannello può essere associato a un titolo con o senza un'icona:

```
void addTab(String title, Component component)
void addTab(String title, Icon icon, Component component, String tip)
void remove(Component component)
void removeAll()
int getTabCount()
```

Per operare sui componenti presenti all'interno dei tab sono disponibili i seguenti metodi;

```
Component getSelectedComponent()
void setSelectedComponent(Component c)
int getSelectedIndex()
void setSelectedIndex(int index)
int indexOfComponent(Component component)
String getTitleAt(int index)
```

La gestione degli eventi su JTabbedPane è abbastanza limitata, dal momento che gli oggetti di tipo ChangeEvent non contengono nessuna informazione sul tipo di evento che li ha generati (in pratica, è impossibile, per un ascoltatore, distinguere tra eventi di selezione, di aggiunta o di rimozione di tab). Per aggiungere o rimuovere un ascoltatore si utilizza la caratteristica coppia di metodi:

```
void addChangeListener(ChangeListener l)
void removeChangeListener(ChangeListener l)
```

Gli ascoltatori di tipo ChangeListener sono caratterizzato dal metodo void stateChanged (ChangeEvent e); gli eventi di tipo ChangeEvent contengono il solo metodo Object getSource(), che permette di ottenere un riferimento all'oggetto che ha generato l'evento. Per conoscere i dettagli dell'evento (numero di Tab, titolo e così via) è necessario interrogare direttamente il componente sorgente.

# Accessori e decorazioni

## JOptionPane

La classe JOptionPane permette di realizzare facilmente finestre modali di input, di allarme o di scelta multipla, ossia quel genere di finestre che vengono utilizzate qualora sia necessario segnalare un malfunzionamento, o presentare all'utente un insieme di scelte su come procedere nell'esecuzione di un programma. L'API JOptionPane mette a disposizione tre tipi di pannelli:

confirm dialog, input dialog e message dialog. Il primo tipo di pannello viene usato quando si deve chiedere all'utente di effettuare una scelta tra un ventaglio di possibilità; il secondo torna utile per richiedere l'inserimento di una stringa di testo; il terzo, infine, viene usato per informare l'utente di un evento. La classe JOptionPane fornisce un gruppo di metodi statici che permettono di creare facilmente questi pannelli ricorrendo a una sola riga di codice. Ecco un esempio di confirm dialog:

```
JOptionPane.showConfirmDialog(null, "Salvare le modifiche?");
```

e uno di message dialog:

```
JOptionPane.showMessageDialog(null,
 "Questo programma ha eseguito un'operazione non valida e sarà terminato...", "Errore", JOptionPane.ERROR_MESSAGE);
```

**Figura 15.4 –** *Un pannello di conferma può aiutare a evitare guai.*

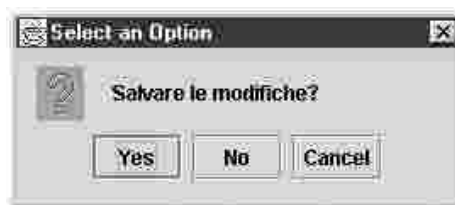
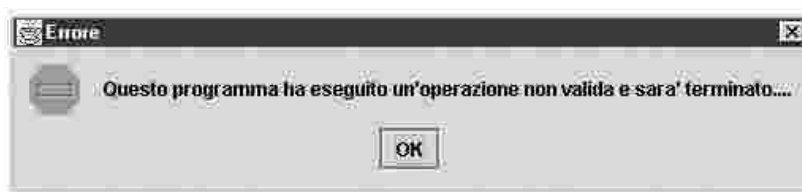



**Figura 15.5 –** *Un minaccioso pannello di notifica annuncia che ormai è troppo tardi.*

Come si può vedere, non è stato necessario creare esplicitamente alcun oggetto di tipo JOptionPane: in entrambi i casi è bastato richiamare un metodo statico che ha provveduto a creare un oggetto grafico con le caratteristiche specificate dai parametri. In questo paragrafo ci si concentrerà sull'utilizzo di un sottoinsieme di tali metodi, che permettono di affrontare la stragrande maggioranza delle situazioni in modo compatto ed elegante:

```
static int showConfirmDialog(Component parentComponent, Object message)
static int showConfirmDialog(Component
```

```
 parentComponent, Object message, String title, int optionType, int messageType)

static String showInputDialog(Component parentComponent, Object message)
static String showInputDialog(Component parentComponent, Object message, String title, int messageType)

static void showMessageDialog(Component parentComponent, Object message)
static void showMessageDialog(Component parentComponent, Object message, String title, int messageType)
```

Le finestre di conferma permettono di scegliere tra un gruppo di opzioni (tipicamente Yes, No e Cancel), mentre le finestre di input servono a inserire del testo; infine, i message dialog si usano per informare l'utente di una particolare situazione. Per ognuna di queste situazioni sono disponibili una chiamata generica e una che permette invece di specificare un gran numero di parametri. Vediamo quali sono i più importanti:

Parent Component

Questo parametro serve a specificare il frame principale. Esso verrà bloccato fino al termine dell'interazione. Ponendo a null questo parametro, la finestra verrà visualizzata al centro dello schermo e risulterà indipendente dal resto dell'applicazione.

Message

Questo campo permette di specificare una stringa da visualizzare come messaggio. In alternativa a String, si può passare una Icon o una qualsiasi sottoclasse di Component.

OptionType

I confirm dialog possono presentare diversi gruppi di opzioni a seconda del valore di questo parametro. Esso può assumere i seguenti valori costanti:

```
JOptionPane.YES_NO_OPTION
JOptionPane.YES_NO_CANCEL_OPTION
JOptionPane.OK_CANCEL_OPTION
```

Message Type

Mediante questo parametro è possibile influenzare l'aspetto complessivo della finestra, per quanto attiene al tipo di icona, al titolo e al layout. Il parametro può assumere uno dei seguenti valori:

```
JOptionPane.ERROR_MESSAGE
JOptionPane.INFORMATION_MESSAGE
JOptionPane.WARNING_MESSAGE
JOption-Pane.QUESTION_MESSAGE
JOptionPane.PLAIN_MESSAGE
```

Le finestre create con i metodi showConfirmDialog e showMessageDialog restituiscono un intero che fornisce informazioni su quale scelta è stata effettuata dall'utente. Esso può assumere uno dei seguenti valori:

```
JOptionPane.YES_OPTION
JOptionPane.NO_OPTION
JOptionPane.CANCEL_OPTION
JOption-Pane.OK_OPTION
JOptionPane.CLOSED_OPTION
```

Nel caso di showInputDialog() viene invece restituita una stringa di testo, o null se l'utente ha annullato l'operazione. Il programmatore può prendere decisioni sul modo in cui proseguire leggendo e interpretando la risposta in maniera simile a come si vede in questo esempio:

```
int returnVal = JOptionPane.showConfirmDialog(null, “Salvare le modifiche?”);
if(returnVal == JOptionPane.YES_OPTION)
        // procedura da eseguire in caso affermativo
else if(returnVal == JOptionPane.NO_OPTION)
        // procedura da eseguire in caso negativo
else;
        // operazione abortita
```

## JFileChooser

Un file chooser è un oggetto grafico che permette di navigare il file system e di selezionare uno o più file su cui eseguire una determinata operazione. Qualsiasi applicazione grafica ne utilizza uno per facilitare le operazioni su disco. JFileChooser offre questa funzionalità tramite un'accessoriata finestra modale.

Si può creare un'istanza di JFileChooser utilizzando i seguenti costruttori:

```
JFileChooser()
JFileChooser(File currentDirectory)
```

Crea un JFileChooser che punta alla directory specificata dal parametro. Per visualizzare un JFileChooser è possibile ricorrere alla seguente coppia di metodi, a seconda che si desideri aprire una finestra di apertura o di salvataggio file:

```
int showOpenDialog(Component parent)
int showSaveDialog(Component parent)
```

Entrambi i metodi restituiscono un intero che può assumere uno dei tre valori:

```
JFile-Chooser.CANCEL_OPTION
JFileChooser.APPROVE_OPTION
JFile-Chooser.ERROR_OPTION
```

**Figura 15.6 –** *Un esempio di JFileChooser.*

Il programmatore può decidere cosa fare dei file selezionati basandosi su queste risposte. Come di consueto per le finestre modali, entrambi i metodi richiedono un reference a un componente, in modo da bloccare il JFrame principale per tutta la durata dell'operazione. Passando null come parametro, il JFrame verrà visualizzato al centro dello schermo e risulterà indipendente dalle altre finestre. Per conoscere il risultato dell'interrogazione, è possibile usare i seguenti metodi:

```
File getCurrentDirectory()
File getSelectedFile()
```

Tali parametri possono essere impostati per via programmatica attraverso la seguente coppia di metodi:

```
void setCurrentDirectory(File dir)
void setSelectedFile(File file)
```

Alcuni metodi consentono un uso più avanzato di JFileChooser. Il metodo setDialogTitle(String dialogTitle) permette di impostare il titolo del JFileChooser; setFileSelectionMode(int mode) consente di abilitare il JFileChooser a selezionare solo file, solo directory o entrambi a seconda del valore del parametro:

```
JFileChooser.FILES_ONLY
JFileChooser.DIRECTORIES_ONLY
JFileChooser.FILES_AND_DIRECTORIES
```

Il metodo setMultiSelectionEnabled(boolean b) abilita o disabilita la possibilità di selezionare più di un file per volta. In questa modalità, per interrogare o modificare lo stato del componente si ricorrerà ai due metodi che seguono:

```
void setSelectedFiles(File[] selectedFiles)
File[] getSelectedFiles()
```

Nelle righe seguenti si può osservare una tipica procedura che fa uso di JFileChooser:

```
class MyFrame extends JFrame {
....
   fileChooser = new JFileChooser();
   int response = fileChooser.showOpenDialog(this);
   if(response == JFileChooser.APPROVE_OPTION) {
      File f = fileChooser.getSelectedFile();
      // qui viene eseguita l'operazione sul file
   }
....
}
```

## Colori e JColorChooser

Un parametro molto importante nei programmi grafici è senza dubbio il colore. Ogni componente grafico Java presenta una quadrupla di metodi che permettono di leggere o impostare i colori di sfondo (background) e di primo piano (foreground):

```
Color getBackground()
void setBackground(Color c)
Color getForeground()
void setForeground(Color c)
```

I componenti Swing usano per default uno sfondo trasparente. Se si desidera impostare un colore di sfondo, bisogna prima rendere opaco il componente con il metodo setOpaque(true).

La classe Color permette di descrivere i colori mediant le tre componenti cromatiche red, green e blue (RGB). Il costruttore principale permette di specificare le tre componenti come interi compresi tra 0 e 255:

```
public Color(int r , int g , int b)
```

I colori più comuni sono disponibili anche sotto forma di costanti della classe Color. Di seguito viene fornito un elenco di tali costanti e il rispettivo valore RGB:

```
public final static Color WHITE        =    new Color(255, 255, 255);    // Bianco
public final static Color LIGHT_GRAY   =    new Color(192, 192, 192);    // Grigio Chiaro
public final static Color GRAY         =    new Color(128, 128, 128);    // Grigio
public final static Color DARK_GRAY    =    new Color(64, 64, 64);       // Grigio Scuro
public final static Color BLACK        =    new Color(0, 0, 0);          // Nero
public final static Color RED          =    new Color(255, 0, 0);        // Rosso
public final static Color PINK         =    new Color(255, 175, 175);    // Rosa
public final static Color ORANGE       =    new Color(255, 200, 0);      // Arancio
public final static Color YELLOW       =    new Color(255, 255, 0);      // Giallo
public final static Color GREEN        =    new Color(0, 255, 0);        // Verde
public final static Color MAGENTA      =    new Color(255, 0, 255);      // Magenta
public final static Color CYAN         =    new Color(0, 255, 255);      // Ciano
public final static Color BLUE         =    new Color(0, 0, 255);        // Blu
```

Le costanti appena illustrate sono presenti nelle versioni del JDK a partire dalla 1.4. Le versioni precedenti dispongono di un insieme di costanti equivalenti, con il nome in lettere minuscole e convenzione CamelCase (per esempio Color.white, Color.lightGray e così via). Tali costanti, ora deprecated, sono disponibili anche nelle versioni del JDK più recenti.

Il pacakge Swing dispone un color chooser, un componente che permette di navigare i colori di sistema con una pratica interfaccia grafica e di selezionarne uno. JColorChooser può essere utilizzato sia come componente separato sia come finestra di dialogo. Nel primo caso bisogna creare un JColorChooser e inserirlo all'interno dell'interfaccia grafica come un qualsiasi altro componente. I metodi void setColor(Color c) e Color getColor() permettono di impostare il colore iniziale o di leggere quello selezionato dall'utente. È anche possibile utilizzare un PropertyChangeListener in modo da essere informati non appena l'utente seleziona un colore dalla palette:

```
JColorChooser c = new JColorChooser();
c.addPropertyChangeListener(new PropertyChangeListener() {
  public void propertyChange(PropertyChangeEvent e) {
    if(e.getPropertyName().equals("color"))
      System.out.println("Hai selezionato il colore " + e.getNewValue());
  }
});
```

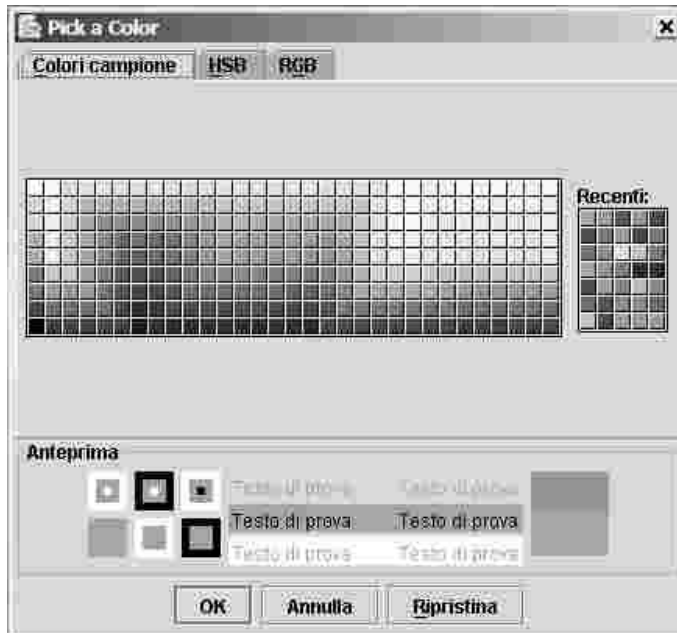
L'uso dei PropertyChangeListener verrà approfondito nel capitolo 18.

Per utilizzare JColorChooser sotto forma di finestra di dialogo, è disponibile un comodo metodo statico, che rende l'utilizzo del componente particolarmente rapido:

```
static Color showDialog(Component component, String title, Color initialColor)
```

In questo caso il color chooser viene visualizzato all'interno di una finestra modale che scompare non appena l'utente ha effettuato una scelta. Come parametri è necessario specificare la

finestra da bloccare, il titolo da dare alla finestra e il colore su cui impostare il color chooser al momento della visualizzazione. L'oggetto restituito corrisponde al colore selezionato dall'utente. Se l'utente ha annullato l'operazione, invece, viene restituito null.

**Figura 15.7 –** *JColorChooser permette di selezionare un colore utilizzando tre tipi di palette diverse.*

## Font e FontChooser

La classe Font incapsula tutte le informazioni relative ai font della piattaforma ospite. Il costruttore permette di specificare il nome di un font, il suo stile e la sua dimensione in punti tipografici:

```
Font(String name, int style, int size)
```

Il parametro style può assumere quattro valori, a seconda che si desideri un carattere normale, grassetto, corsivo o grassetto e corsivo insieme:

```
Font.PLAIN
Font.BOLD
Font.ITALIC
Font.BOLD | Font.ITALIC
```

Il nome del font può essere di due tipi: logico o fisico. Il nome logico può assumere i seguenti valori: Serif, Sans-serif, Monospaced, Dialog e DialogInput. Tutte le piattaforme che supportano Java forniscono un mapping tra questi nomi logici e i font reali presenti sul sistema, in modo da permettere al programmatore di scrivere applicazioni portabili. Il nome fisico, invece, identifica un preciso font di sistema, e per questo un font valido su una particolare macchina può non esistere su un'altra. A partire dal JDK 1.2, la distribuzione standard di Java comprende comunque i seguenti font fisici:

```
LucidaBrightDemiBold.ttf
LucidaBrightDemiItalic.ttf
LucidaBrightItalic.ttf
LucidaBrightRegular.ttf
LucidaSansDemiBold.ttf
LucidaSansDemiOblique.ttf
LucidaSansOblique.ttf
LucidaSansRegular.ttf
LucidaTypewriterBold.ttf
LucidaTypewriterBoldOblique.ttf
LucidaTypewriterOblique.ttf
LucidaTypewriterRegular.ttf
```

Esiste un sistema per conoscere il nome fisico di tutti i font presenti nel sistema. Si tratta dell'oggetto `GraphicEnvironment`, accessibile tramite il metodo statico:

```
GraphicEnvironment.getLocalGraphicsEnvironment()
```

Esso dispone di un metodo statico in grado di restituire l'elenco dei font installati:

```
Font[] getAllFonts()
```

Grazie a questa chiamata è possibile scrivere con poche righe un programma che stampi a console i nomi di tutti i font di sistema:

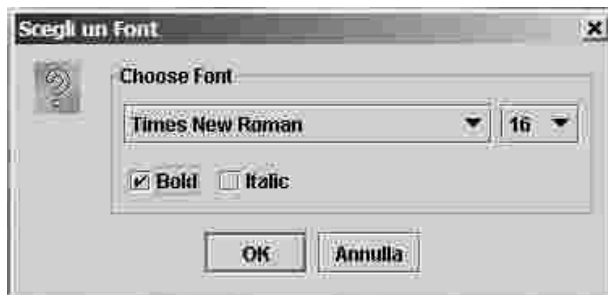
```
import java.awt.*;

public class FontEstractor {
  public static void main(String argv[]) {
    Font[] fonts = GraphicsEnvironment.getLocalGraphicsEnvironment().getAllFonts();
    for ( int i = 0; i < fonts.length; i++)
      System.out.println(fonts[i].toString());
  }
}
```

Dal momento che il package Swing non contiene un componente `JFontChooser`, verrà ora mostrato come crearne uno. Si definisce una classe `JFontChooser`, sottoclasse di `JComponent`,

provvista di due JComboBox e due JCheckBox che permettono di selezionare un font, la sua dimensione e lo stile. Questo programma è la dimostrazione di come sia semplice creare dal niente un nuovo componente grafico da utilizzare in numerosi contesti:

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.beans.*;

public class FontChooser extends JComponent {

  private JComboBox fontNameBox;
  private JComboBox fontSizeBox;
  private JCheckBox boldCheckBox;
  private JCheckBox italicCheckBox;

  public FontChooser() {
    fontNameBox = new JComboBox();
    fontSizeBox = new JComboBox();
    boldCheckBox = new JCheckBox("Bold", false);
    italicCheckBox = new JCheckBox("Italic", false);

    Font[] fonts = GraphicsEnvironment.getLocalGraphicsEnvironment().getAllFonts();
    for ( int i = 0; i < fonts.length; i++)
      fontNameBox.addItem(fonts[i].getName());
    for ( int i = 6; i < 200; i++)
      fontSizeBox.addItem(new Integer(i));

    fontSizeBox.setSelectedIndex(12);
    setLayout(new GridLayout(0, 1));

    JPanel comboBoxPanel = new JPanel();
      comboBoxPanel.setLayout(new FlowLayout(FlowLayout.LEFT));
      comboBoxPanel.add(fontNameBox);
      comboBoxPanel.add(fontSizeBox);
    add(comboBoxPanel);

    JPanel checkBoxPanel = new JPanel();
      checkBoxPanel.setLayout(new FlowLayout(FlowLayout.LEFT));
      checkBoxPanel.add(boldCheckBox);
      checkBoxPanel.add(italicCheckBox);
    add(checkBoxPanel);
    setBorder(BorderFactory.createTitledBorder("Choose Font"));
    ActionListener eventForwarder = new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        setFont(new Font((String)fontNameBox.getSelectedItem(), (boldCheckBox.isSelected()
```

```
? Font.BOLD : Font.PLAIN) | (italicCheckBox.isSelected()
? Font.ITALIC : Font.PLAIN), ((Integer)fontSizeBox.getSelectedItem()).intValue()));
    }
  };
  fontNameBox.addActionListener(eventForwarder);
  fontSizeBox.addActionListener(eventForwarder);
  boldCheckBox.addActionListener(eventForwarder);
  italicCheckBox.addActionListener(eventForwarder);
}

public void setFont(Font f) {
  super.setFont(f);  // Questa chiamata genera un PropertyChangeEvent
  fontNameBox.setSelectedItem(f.getName());
  fontSizeBox.setSelectedItem(new Integer(f.getSize()));
  boldCheckBox.setSelected(f.isBold());
  italicCheckBox.setSelected(f.isItalic());
}
public static Font showFontChooser(Component parent,String title , Font initialFont) {
  FontChooser fc = new FontChooser();
  fc.setFont(initialFont);
  int answer = JOptionPane.showConfirmDialog(parent, fc, title, JOptionPane.OK_CANCEL_OPTION);
  if(answer != JOptionPane.OK_OPTION)
    return null;
  else
    return fc.getFont();
}
}
```

**Figura 15.8 –** *Utilizzando i concetti esposti finora, è possibile creare un pratico JFontChooser.*

JFontChooser presenta due modalità d'uso del tutto simili a quelle di JColorChooser: da una parte è possibile crearlo come componente separato da usare all'interno di interfacce grafiche, dall'altra è possibile creare una finestra di dialogo mediante il seguente metodo statico:

```
static Font showFontChooser(Component parent , String title , Font initialFont)
```

Tale metodo richiede come parametri il componente parent, il titolo della finestra e il font con il quale impostare il componente al momento dell'apertura. Il valore di ritorno riporta il font selezionato o null se l'utente ha annullato l'operazione:

```
Font f = showFontChooser(null,"Scegli un Font",new Font("monospaced",0,15));
if(f != null) {
  JLabel label = new JLabel("Test");
  label.setFont(f);
  JOptionPane.showMessageDialog(null,label);
}
```

Anche questo componente genera un PropertyChangeEvent ogni volta che l'utente seleziona un nuovo stile. Per gestire l'evento si può utilizzare un frammento di codice del tipo:

```
FontChooser fc = new FontChooser("FontChooser");
final JLabel label = new JLabel("Test");
fc.setFont(new Font("Times New Roman", 3, 120));
fc.addPropertyChangeListener(new PropertyChangeListener() {
  public void propertyChange(PropertyChangeEvent e) {
    if(e.getPropertyName().equals("font"))
      label.setFont((Font)e.getNewValue());
  }
});
```

# Pluggable look & feel

Ogni ambiente a finestre è caratterizzato da due proprietà fondamentali: l'aspetto dei componenti (ossia la loro sintassi), e la maniera in cui essi reagiscono alle azioni degli utenti (la loro semantica). L'insieme di queste proprietà viene comunemente definito look & feel.

Chiunque abbia provato a lavorare con un sistema Linux dopo anni di pratica su piattaforma Windows si sarà reso conto di quanto sia difficile abituarsi a una nuova semantica: le mani tendono a comportarsi come sulla vecchia piattaforma, ma la reazione che osserviamo con gli occhi non è quella che ci aspettavamo. Per esempio, in Linux è normale che le finestre si espandano verticalmente invece che a pieno schermo, e che i menu scompaiano quando si rilascia il pulsante del mouse.

La natura multi piattaforma di Java ha spinto i progettisti di Swing a separare le problematiche di disegno grafico dei componenti da quelle inerenti al loro contenuto informativo, con la sorprendente conseguenza di permettere agli utenti di considerare il look & feel come una proprietà del componente da impostare a piacere. La distribuzione standard del JDK comprende di base due alternative: Metal e Motif. La prima definisce un look & feel multipiattaforma, progettato per risultare il più possibile familiare a chiunque. La seconda implementa una vista

familiare agli utenti Unix. Le distribuzioni di Java per Windows e Mac includono anche un look & feel che richiama quello della piattaforma ospite. Per motivi di copyright Sun non può proporre queste due scelte su piattaforme diverse. Alcune software house indipendenti distribuiscono, sotto forma di file JAR, dei package contenenti dei look & feel alternativi, che è possibile aggiungere alla lista dei look & feel di sistema.

Per impostare da programma un particolare look & feel, è sufficiente chiamare il metodo UIManager.setLookAndFeel(String className) passando come parametro il nome di un look & feel nstallato nel sistema. Quindi, è necessario chiamare il metodo statico updateComponentTreeUI(Component c) della classe SwingUtilities sulla finestra principale, in modo da forzare tutti i componenti dell'interfaccia ad aggiornare il proprio look & feel. Per esempio:

```
try {
  UIManager.setLookAndFeel("com.sun.java.swing.plaf.motif.MotifLookAndFeel");
  SwingUtilities.updateComponentTreeUI(frame);
}
catch (Exception e) {}
```

Di seguito si presentano le stringhe relative ai quattro look & feel descritti sopra:

```
"javax.swing.plaf.metal.MetalLookAndFeel"
"com.sun.java.swing.plaf.windows.WindowsLookAndFeel"
"com.sun.java.swing.plaf.motif.MotifLookAndFeel"
"javax.swing.plaf.mac.MacLookAndFeel"
```

L'ultima definisce il look & feel Mac, disponibile solo su piattaforma Apple.

Se si desidera interrogare il sistema per conoscere il nome e la quantità dei look & feel installati, si può ricorrere ai seguenti metodi statici di UIManager:

```
static String getSystemLookAndFeelClassName()
```

Restituisce il nome del look & feel che implementa il sistema a finestre della piattaforma ospite (Windows su sistemi Microsoft, Mac su macchine Apple e Motif su piattaforma Solaris). Se non esiste una scelta predefinita, viene restituito il nome del look & feel Metal.

```
static String getCrossPlatformLookAndFeelClassName()
```

Restituisce il nome del look & feel multipiattaforma: il Java look & feel (JLF).

```
static UIManager.LookAndFeelInfo[] getInstalledLookAndFeels()
```

Restituisce un vettore di oggetti che forniscono alcune informazioni sui look & feel installati nel sistema, come per esempio il nome (accessibile con il metodo getName()).

Il seguente esempio crea una finestra con tanti JRadioButton quanti sono i look & feel dispo-

nibili nel sistema. Ogni volta che l'utente preme uno dei pulsanti, il look & feel viene aggiornato di conseguenza:

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class LookAndFeelExample extends JFrame {
  public LookAndFeelExample() {
    super("LookAndFeelExample");
    getContentPane().setLayout(new GridLayout(0, 1));
    ButtonGroup group = new ButtonGroup();
    ActionListener buttonListener = new ButtonListener();
    getContentPane().add(new JLabel("Scegli un Look & Feel"));

    UIManager.LookAndFeelInfo[] lookAndFeelList = UIManager.getInstalledLookAndFeels();
    for ( int i = 0; i < lookAndFeelList.length; i++) {
      JRadioButton b = new JRadioButton(lookAndFeelList[i].getClassName());
      b.addActionListener(buttonListener);
      group.add(b);
      getContentPane().add(b);
    }
    pack();
  }
  public void changeLookAndFeel(String s) {
    try {
      UIManager.setLookAndFeel(s);
      SwingUtilities.updateComponentTreeUI(this);
    }
    catch (Exception ex) {}
  }
  class ButtonListener implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      JRadioButton b = (JRadioButton)e.getSource();
      changeLookAndFeel(b.getText());
    }
  }

  public static void main(String argv[]) {
    LookAndFeelExample e = new LookAndFeelExample();
    e.setVisible(true);
  }
}
```

**Figura 15.9 –** *Un programma di esempio visto con tre look & feel diversi.*

# Border

Una caratteristica che Swing offre in esclusiva è la possibilità di assegnare un bordo diverso a ogni singolo componente grafico, sia esso un pannello, un pulsante o una Tool Bar. Per aggiungere un bordo a un componente, è sufficiente chiamare il metodo setBorder(Border b), passando come parametro un'istanza di una qualsiasi delle classi descritte di seguito.

Il package javax.swing.border offre ben sette tipi di bordo, il più semplice dei quali è composto da una singola riga dello spessore specificato.

```
LineBorder(Color color, int thickness, boolean roundedCorners)
```

Il tipo mostrato sopra crea un bordo a linea, con il colore, lo spessore e il tipo di bordo specificati. I bordi seguenti, invece, ricreano effetti tridimensionali. Essi richiedono come parametro un intero che può assumere il valore BevelBorder.LOWERED o BevelBorder.RAISED, a seconda che si desideri un effetto in rilievo o rientrante:

```
BevelBorder(int bevelType)
```

Crea un bordo in rilievo, del tipo specificato dal parametro.

SoftBevelBorder(int bevelType)

Crea un bordo in rilievo sfumato, del tipo specificato dal parametro.

EtchedBorder(int etchType)

Crea un bordo scolpito, del tipo specificato dal parametro.

Qualora si desideri creare attorno a un componente una vera e propria cornice di spessore arbitrario, è possibile ricorrere ai seguenti oggetti, che permettono creare bordi vuoti, a tinta unita o decorati con un'immagine GIF o JPEG:

EmptyBorder(int top, int left, int bottom, int right)

Crea un bordo vuoto dello spessore specificato.

MatteBorder(Icon tileIcon)

Crea un bordo utilizzando un'immagine.

MatteBorder(int top, int left, int bottom, int right, Icon tileIcon)

Crea un bordo delle dimensioni specificate, utilizzando un'icona.

MatteBorder(int top, int left, int bottom, int right, Color matteColor)

Crea un bordo delle dimensioni e del colore specificati.

Per finire, è disponibile una coppia di bordi che permette di creare composizioni a partire da altri bordi:

TitledBorder(Border border, String title, int titleJustification, int titlePosition, Font titleFont, Color titleColor)

Crea una cornice, composta dal bordo che viene passato come primo parametro e dal titolo specificato dal secondo parametro. Il terzo parametro può assumere i valori TitledBorder.CENTER, TitledBorder.LEFT o TitledBorder.RIGHT. Il quarto, che specifica la posizione del titolo, può assumere invece i valori Titled-Border.ABOVE_BOTTOM, TitledBorder.ABOVE_TOP, Titled-Border.BELOW_BOTTOM, TitledBorder.BELOW_TOP. Gli ultimi due parametri specificano font e colore del titolo. Sono disponibili anche costruttori più semplici, per esempio uno che richiede solo i primi due parametri e uno che omette gli ultimi due.

CompoundBorder(Border outsideBorder, Border insideBorder)

Crea una cornice componendo i due bordi passati come parametro.

# Un'applicazione grafica complessa

Le nozioni apprese finora permettono di affrontare lo studio di un'applicazione grafica di una discreta complessità. Le seguenti righe permettono di realizzare un piccolo editor di testo perfettamente funzionante, utilizzando una JTextArea, una JToolBar, una JMenuBar e un JFileChooser, e mostrando un utilizzo pratico delle Action. Viene inoltre illustrato, all'interno dei metodi loadText() e saveText(), come sia possibile inizializzare un JTextComponent a partire da un file su disco.

```
import javax.swing.*;
import javax.swing.text.*;
import java.awt.*;
import java.awt.event.*;
import java.io.*;

public class TextEditor extends JFrame {
  private JTextComponent editor;
  private JFileChooser fileChooser;
  protected Action loadAction;
  protected Action saveAction;
  protected Action cutAction;
  protected Action copyAction;
  protected Action pasteAction;
  public TextEditor() {
    super("TextEditor");
    setSize(300, 300);
    createActions();
    JMenuBar menuBar = createMenuBar();
    JToolBar toolBar = createToolBar();
    editor = createEditor();
    JComponent centerPanel = createCenterComponent();
    getContentPane().add(BorderLayout.NORTH, toolBar);
    getContentPane().add(BorderLayout.CENTER, centerPanel);
    setJMenuBar(menuBar);
    fileChooser = new JFileChooser();
    setVisible(true);
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
  }
  protected void createActions() {
    loadAction = new AbstractAction("Open", new ImageIcon("Open24.gif")) {
      public void actionPerformed(ActionEvent e) {
        loadText();
      }
    };
    saveAction = new AbstractAction("Save", new ImageIcon("Save24.gif")) {
      public void actionPerformed(ActionEvent e) {
        saveText();
```

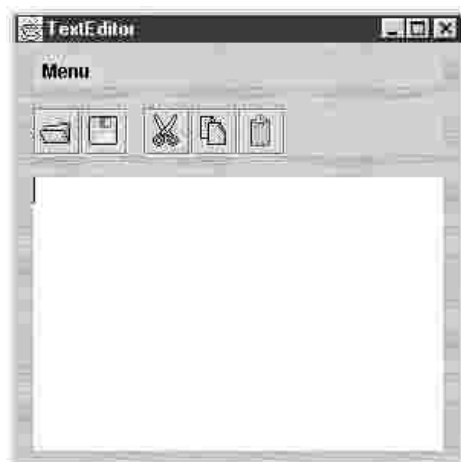
```
      }
    };
    cutAction = new AbstractAction("Cut", new ImageIcon("Cut24.gif")) {
      public void actionPerformed(ActionEvent e) {
        editor.cut();
      }
    };
    copyAction = new AbstractAction("Copy", new ImageIcon("Copy24.gif")) {
      public void actionPerformed(ActionEvent e) {
        editor.copy();
      }
    };
    pasteAction = new AbstractAction("Paste", new ImageIcon("Paste24.gif")) {
      public void actionPerformed(ActionEvent e) {
        editor.paste();
      }
    };
  }
  protected JToolBar createToolBar() {
    JToolBar tb = new JToolBar();
    tb.add(loadAction);
    tb.add(saveAction);
    tb.addSeparator();
    tb.add(cutAction);
    tb.add(copyAction);
    tb.add(pasteAction);
    return tb;
  }
  protected JMenuBar createMenuBar() {
    JMenu menu = new JMenu("Menu");
    menu.add(loadAction);
    menu.add(saveAction);
    menu.addSeparator();
    menu.add(cutAction);
    menu.add(copyAction);
    menu.add(pasteAction);
    JMenuBar menuBar = new JMenuBar();
    menuBar.add(menu);
    return menuBar;
  }
  protected JComponent createCenterComponent() {
    if(editor == null)
      editor = createEditor();
    return new JScrollPane(editor);
  }
  protected JTextComponent createEditor() {
    return new JTextArea();
  }
```

```
  public void loadText() {
    int response = fileChooser.showOpenDialog(this);
    if(response == JFileChooser.APPROVE_OPTION) {
      try {
        File f = fileChooser.getSelectedFile();
        Reader in = new FileReader(f);
        editor.read(in, null);
        setTitle(f.getName());
      }
      catch(Exception e) {}
    }
  }
  public void saveText() {
    int response = fileChooser.showSaveDialog(this);
    if(response == JFileChooser.APPROVE_OPTION) {
      try {
        File f = fileChooser.getSelectedFile();
        Writer out = new FileWriter(f);
        editor.write(out);
        setTitle(f.getName());
      }
      catch(Exception e) {}
    }
  }
  public static void main(String argv[]) {
    TextEditor t = new TextEditor();
  }
}
```

**Figura 15.10 –** *Con appena un centinaio di righe è possibile realizzare un editor di testi completo.*

Il metodo createActions() riesce a definire cinque classi in appena trenta righe di codice, facendo uso delle classi anonime. L'uso di classi anonime in questo contesto è giustificato dal proposito di rendere il programma molto compatto. Questo programma vuole anche dare una dimostrazione di costruzione modulare di interfacce grafiche. Come si può notare, il costruttore genera gli elementi dell'interfaccia grafica ricorrendo a un gruppo di metodi factory, vale a dire metodi protected caratterizzati dal prefisso create, come createToolBar(), createMenuBar(), createCenterComponent() e createEditor(), i quali restituiscono il componente specificato dal loro nome. Questa scelta offre la possibilità di creare sottoclassi del programma che implementino una differente composizione della GUI semplicemente sovrascrivendo questi metodi, e lasciando inalterato il costruttore. Ridefinendo i metodi factory è possibile modificare in misura evidente l'aspetto dell'applicazione, aggiungendo un bordo alla Menu Bar, alla Tool Bar e al pannello centrale, senza bisogno di alterare il costruttore del programma:

```
import java.awt.*;
import javax.swing.*;
import javax.swing.border.*;

public class BorderedTextEditor extends TextEditor {
  protected JMenuBar createMenuBar() {
    JMenuBar mb = super.createMenuBar();
    mb.setBorder(new MatteBorder(7, 12, 7, 12, new ImageIcon("Texture_wood_004.jpg")));
    mb.setBackground(new Color(224, 195, 96));
    return mb;
  }
    protected JToolBar createToolBar() {
      JToolBar tb = super.createToolBar();
      tb.setBorder(new MatteBorder(7, 12, 7, 12, new ImageIcon("Texture_wood_004.jpg")));
      tb.setBackground(new Color(224, 195, 96));
      return tb;
    }
    protected JComponent createCenterComponent() {
      JComponent c = super.createCenterComponent();
      c.setBorder(new MatteBorder(7, 12, 7, 12, new ImageIcon("Texture_wood_004.jpg")));
      return c;
    }
  public static void main(String argv[]) {
    BorderedTextEditor t = new BorderedTextEditor();
  }
}
```

**Figura 15.11 –** *Sovrascrivendo i metodi factory è possibile modificare il comportamento di un'applicazione senza alterarne la struttura.*

# Il disegno in Java

Andrea Gini

## Il disegno in Java

Dopo aver concluso la panoramica sui principali componenti grafici Java, è giunto il momento di vedere come si utilizzano le primitive di disegno, grazie alle quali è possibile creare componenti ex novo.

Dal punto di vista del programmatore, un componente grafico non è altro che un oggetto al quale viene assegnata un'area di schermo su cui disegnare, e che è in grado di ascoltare gli eventi di mouse o tastiera. In questo capitolo verranno illustrate le primitive di disegno grafico e la gestione di eventi di mouse e tastiera, quindi si vedrà come combinare i concetti appresi per creare componenti interattivi.

## JComponent e il meccanismo di disegno

Il primo passo per creare un componente nuovo è creare una sottoclasse di un componente esistente. I componenti Swing visti nei capitoli precedenti sono tutti sottoclassi di `JComponent`, un componente privo di forma che offre il supporto ai soli eventi di mouse e tastiera: esso si presta perciò a fare da base per la creazione di controlli grafici di qualsiasi tipo.

Il sistema grafico di Java funziona grazie a un meccanismo a call back: ogni sottoclasse di `JComponent` dispone di un metodo `paintComponent(Graphics g)`, che viene chiamato direttamente dal sistema in tutte le circostanze in cui è necessario dipingere il componente sullo schermo. Durante il ciclo di vita di un'applicazione, la necessità di ridisegnare un determinato componente si verifica soprattutto in tre circostanze: in occasione della prima visualizzazione, nel corso di

un ridimensionamento o nel caso in cui l'area del componente sia stata "danneggiata", ossia sia stata coperta momentaneamente da un'altra finestra.

Il metodo paintComponent() non deve mai essere invocato direttamente: per motivi di prestazioni e di architettura, infatti, il refresh dello schermo viene avviato solo in alcune circostanze (non ha senso ridisegnare lo schermo più di 30 volte al secondo, dal momento che l'occhio non è in grado di discernere le differenze). Nei casi in cui l'utente desideri richiamare in modo esplicito il refresh, deve farlo tramite il metodo repaint(). Esso invia al sistema una richiesta di refresh che sarà gestita dal thread che si occupa del disegno, appena possibile. Esistono due versioni significative del metodo repaint:

```
public void repaint()
public void repaint(int x, int y, int width, int height)
```

Il primo richiede che l'intero componente venga ridisegnato; il secondo effettua il repaint unicamente nell'area specificata dai parametri (x e y specificano la posizione dell'angolo in alto a sinistra, mentre width e height sono rispettivamente la larghezza e l'altezza). Questo metodo permette di limitare l'area di disegno al frammento che ha subito modifiche dall'ultimo repaint, una strategia che può avere un impatto significativo sulle performance nel caso di programmi grafici che devono effettuare calcoli molto complessi.

## L'oggetto Graphics

Come illustrato nel paragrafo precedente, per creare un componente nuovo è sufficiente definire una sottoclasse di JComponent e dichiarare al suo interno un metodo caratterizzato dalla firma:

```
public void paintComponent(Graphics g)
```

L'oggetto Graphics, che il metodo paintComponent() riceve come parametro, incapsula l'area in cui il è possibile disegnare. Graphics dispone di metodi di disegno. I più importanti sono quelli che permettono di disegnare linee, cerchi, stringhe e rettangoli:

```
boolean drawImage(Image img, int x, int y, ImageObserver observer)
void drawLine(int x1, int y1, int x2, int y2)
void drawOval(int x, int y, int width, int height)
 void drawRect(int x, int y, int width, int height)
void drawString(String str, int x, int y)
```

È possibile disegnare anche cerchi e rettangoli pieni:

```
void fillOval(int x, int y, int width, int height)
void fillRect(int x, int y, int width, int height)
```

Le coordinate usate come argomento su metodi di un oggetto Graphics sono considera-

te in relazione all'angolo in alto a sinistra del componente da disegnare. La coordinata x, dunque,cresce da sinistra a destra, e la y dall'alto in basso. Ogni JComponent possiede una quadrupla di metodi che permettono di leggere e impostare il colore di primo piano e il font del componente:

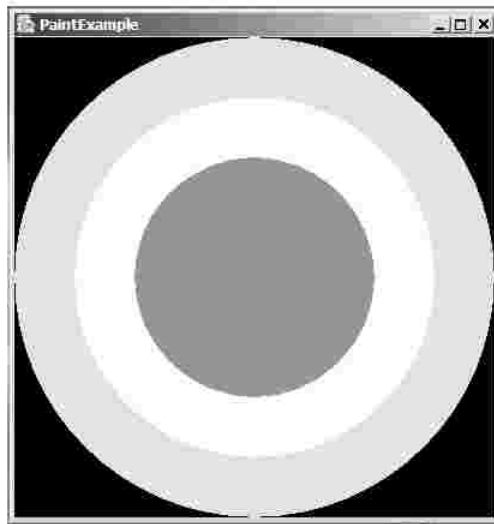
```
Color getForeground()
void setForeground(Color c)
Font getFont()
void setFont(Font f)
```

Al momento di invocare il metodo paintComponent su un determinato componente, il sistema di disegno imposta sull'oggetto Graphics il colore e il font del componente stesso. Durante il disegno, questi parametri possono naturalmente essere modificati, invocando sull'oggetto Graphics i seguenti metodi:

```
void Color getColor()
void setColor(Color c)
Font getFont()
void setFont(Font font)
```

Il prossimo esempio mostra come creare un semplice componente a partire da JComponent. all'interno del metodo paintComponent() sono presenti le direttive per disegna un quadrato nero di 400 x 400 pixel, con all'interno tre cerchi concentrici di colore verde, bianco e rosso:

```
import java.awt.*;
import javax.swing.*;

public class PaintExample extends JComponent {
  public void paintComponent(Graphics g) {
    g.setColor(Color.BLACK);
    g.fillRect(0,0,400,400);
    g.setColor(Color.GREEN);
    g.fillOval(0,0,400,400);
    g.setColor(Color.WHITE);
    g.fillOval(50,50,300,300);
    g.setColor(Color.RED);
    g.fillOval(100,100,200,200);
  }

  public static void main(String argv[]) {
    JFrame f = new JFrame("PaintExample");
    f.setSize(410,430);
    f.getContentPane().add(new PaintExample());
    f.setVisible(true);
  }
}
```

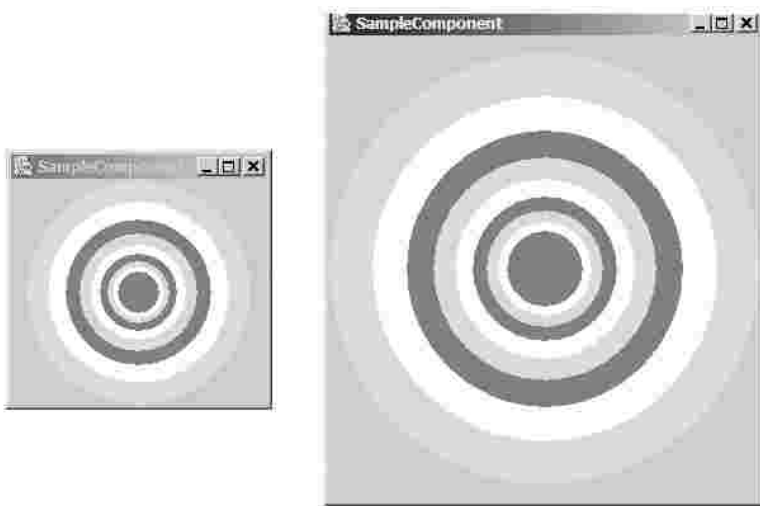
**Figura 16.1 –** *Un primo esempio di componente grafico personalizzato.*

I metodi di disegno vengono eseguiti in sequenza. Quando una direttiva grafica viene eseguita, il suo disegno si sovrappone a quanto eventualmente già presente sullo schermo. Il metodo setColor() modifica il colore del pennello, pertanto esso ha effetto su tutte le istruzioni successive. Il componente PaintExample può essere inserito all'interno delle interfacce grafiche come qualsiasi altro componente. Per creare componenti di reale utilità, è necessario imparare a gestire le circostanze in cui il componente deve essere ridimensionato.

## Adattare il disegno alle dimensioni del clip

Quando si disegna un componente è bene tener conto delle dimensioni del clip, in modo da adattare il disegno di conseguenza. Il metodo getSize(), presente in tutti i componenti grafici Java, restituisce un oggetto di tipo Dimension, che a sua volta possiede come attributi pubblici width e height, pari rispettivamente alla larghezza e all'altezza dell'area di disegno. Conoscendo queste misure, è possibile creare un disegno che sia proporzionale alla superficie da riempire. Nel prossimo esempio, viene creato un componente al cui interno vengono dipinti una serie di cerchi concentrici di colore rosso, bianco e verde. Ridimensionando il componente, il disegno viene ricalcolato in modo da adattarsi alle nuove dimensioni:

```
import java.awt.*;
import javax.swing.*;

public class SampleComponent extends JComponent {
```

```
    private Color[] colors = {Color.RED,Color.GREEN,Color.WHITE};

    public void paintComponent(Graphics g) {
      // Calcola il diametro a partire dalle dimensioni del componente
      Dimension size = getSize();
      int d = Math.min(size.width, size.height);
      // Disegna una serie di cerchi concentrici
      for ( int i = 1; i <10 ; i++) {
         // sceglie a rotazione il colore
        g.setColor(colors[i%3]);
        // Calcola le coordinate del cerchio
        int x = (size.width - d) / 2;
        int y = (size.height - d) / 2;
        g.fillOval(x, y, d,d);
        // Riduce le dimensioni del diametro
        d = d - (d / 10 * 2);
      }
    }
    public static void main(String argv[]) {
      JFrame f = new JFrame();
      f.setSize(500, 300);
      f.getContentPane().add(new SampleComponent());
      f.setVisible(true);
    }
  }
```

**Figura 16.2 –** *Un componente grafico capace di adattarsi al cambiamento delle dimensioni.*

## Disegnare immagini

L'oggetto Graphics dispone anche di un metodo drawImage(), che permette di disegnare immagini GIF o JPEG presenti su disco. Il metodo drawImage() ha diversi formati. I due più importanti sono i seguenti:

```
boolean drawImage(Image img, int x, int y, ImageObserver observer)
boolean drawImage(Image img, int x, int y, int width, int height, ImageObserver observer)
```

Il primo disegna l'immagine a partire dalle coordinate specificate con i parametri x e y, rispettandone le dimensioni originali. Il secondo permette anche di indicare un'area, alla quale l'immagine verrà adattata aumentandone o riducendone le dimensioni. Entrambi i metodi richiedono come parametro un ImageObserver, ossia un oggetto cui viene notificato in modo asincrono il progredire del caricamento dell'immagine. Tutti i componenti grafici Java implementano l'interfaccia ImageObserver, pertanto all'interno del metodo paint questo parametro può essere tranquillamente impostato a this. Per caricare un'immagine da disco, è possibile utilizzare una chiamata di questo tipo, fornendo come parametro l'indirizzo dell'immagine da caricare:

```
Image image = Toolkit.getDefaultToolkit().getImage(url);
```

Il prossimo esempio crea un componente che carica un'immagine e la ridimensiona in modo da coprire tutta l'area disponibile: se si cambiano le dimensioni del frame, l'immagine verrà a sua volta ridimensionata. Per lanciare il programma è necessario specificare da riga di comando l'indirizzo di un'immagine. Per esempio, la chiamata seguente:

```
java DrawImageExample c:\paperino.gif
```

avvia il programma caricando l'immagine paperino.gif dalla radice del disco C:

```
import java.awt.*;
import javax.swing.*;

public class DrawImageExample extends JComponent  {
  private Image image;

  public DrawImageExample(String location) {
    image = Toolkit.getDefaultToolkit().getImage(location);
  }

  public void paintComponent(Graphics g) {
    Dimension size = getSize();
    g.drawImage(image, 0, 0, size.width,size.height,this);
  }

  public static void main(String argv[]) {
```

```
      if(argv.length != 1)
        throw new IllegalArgumentException("Use: java PaintImageExample <image>");
      JFrame f = new JFrame("DrawImageExample");
      f.setSize(600, 500);
      f.getContentPane().add(new DrawImageExample(argv[0]));
      f.setVisible(true);
    }
  }
```

**Figura 16.3 –** *Una dimostrazione del metodo drawImage().*

## Disegnare il testo

L'oggetto Graphics contiene anche metodi per disegnare stringhe di testo:

```
void drawString(String str, int x, int y)
```

Questo metodo richiede semplicemente una stringa e le coordinate del punto in cui disegnarla. Posizionare il testo all'interno di un componente non  certo un compito facile, visto anche che con la maggior parte dei font le dimensioni dei caratteri cambiano a seconda del carattere da visualizzare

(una “i” occupa meno spazio di una “O”). Per permettere un posizionamento agevole, possibile utilizzare l'oggetto FontMetrics, che può essere ricavato direttamente da Graphics grazie al metodo:

```
FontMetrics getFontMetrics()
FontMetrics getFontMetrics(Font f)
```

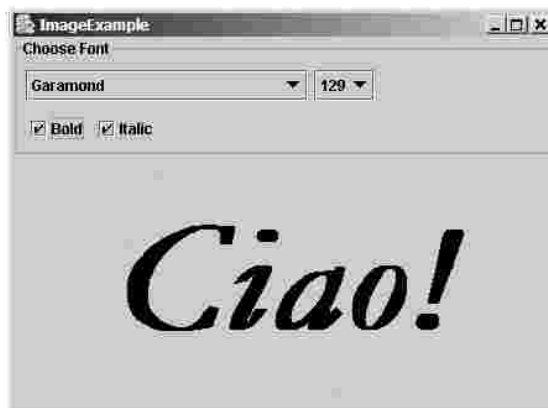
Il primo di questi metodi restituisce il FontMetrics relativo al font attualmente in uso all'interno dell'oggetto Graphics, mentre il secondo permette di ottenere quello di un qualsiasi font di sistema. Grazie a FontMetrics possibile conoscere i principali parametri tipografici del font:

```
int charWidth(char ch)
int getAscent()
int getDescent()
int getHeight()
int getLeading()
int getMaxAdvance()
int getMaxAscent()
int[] getWidths()
boolean hasUniformLineMetrics()
int stringWidth(String str)
```

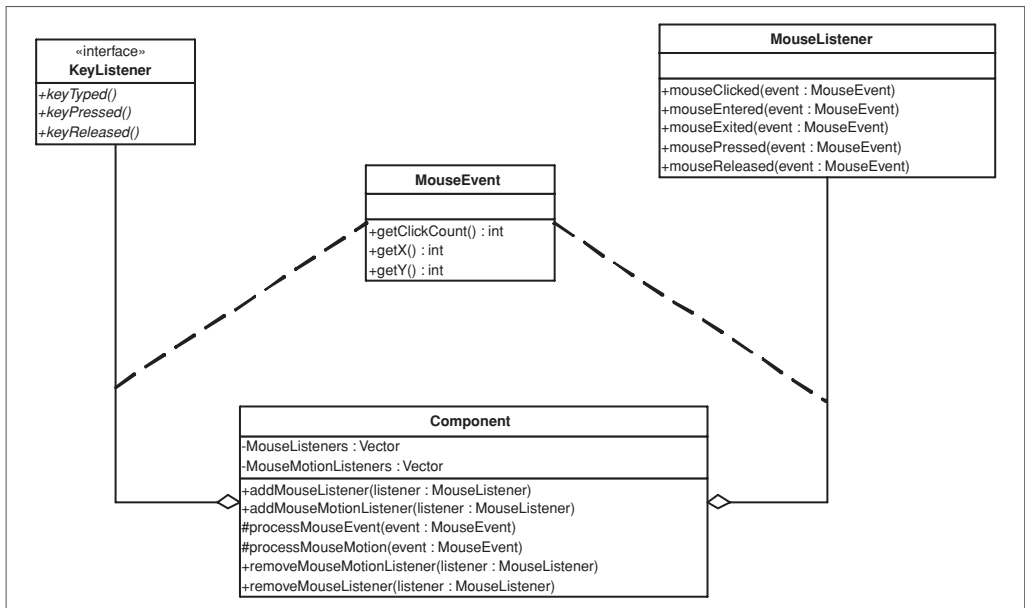
La conoscenza di questi parametri consente di ottenere un controllo pressoché totale sul modo di visualizzare i caratteri a schermo. Ai fini di un uso normale, i due metodi più importanti sono int stringWidth(String s) e int getHeight(), che restituiscono rispettivamente la larghezza in pixel di una determinata stringa in quel font e l’altezza del font stesso (intesa come distanza tra l’estremità inferiore di un carattere discendente, come la p minuscola, e l’estremità superiore dei caratteri più alti, come la I maiuscola ). Il seguente esempio mostra come si possano utilizzare queste misure per disporre una stringa al centro di un pannello. All’interno di questo programma viene utilizzato il componente JFontChooser, il cui sorgente è stato descritto nel capitolo 15.

```
import java.awt.*;
import java.beans.*;
import javax.swing.*;

public class DrawStringExample extends JComponent {
  private String text;

  public DrawStringExample(String text) {
    this.text = text;
  }

  public void paintComponent(Graphics g) {
    FontMetrics metrics = g.getFontMetrics();
    Dimension size = getSize();
    int fontHeight = metrics.getHeight();
    int textWidth = metrics.stringWidth(text);
```

```
    int x = (size.width - textWidth) / 2;
    int y = (size.height + (fontHeight/2)) / 2;
    g.drawString(text,x,y);
  }

  public static void main(String argv[]) {
    if(argv.length != 1)
      throw new IllegalArgumentException("Use: java PaintImageExample <string>");
    JFrame f = new JFrame("ImageExample");
    f.getContentPane().setLayout(new BorderLayout());

    FontChooser fc = new FontChooser();
    final DrawStringExample dse = new DrawStringExample(argv[0]);

    fc.addPropertyChangeListener(new PropertyChangeListener() {
      public void propertyChange(PropertyChangeEvent e) {
        if(e.getPropertyName().equals("font"))
         dse.setFont((Font)e.getNewValue());
         dse.repaint();
      }
    });
    f.setSize(600, 500);
    f.getContentPane().add(BorderLayout.NORTH,fc);
    f.getContentPane().add(BorderLayout.CENTER,dse);

    fc.setFont(dse.getFont());
    f.setVisible(true);
  }
}
```

**Figura 16.4 –** *FontMetrics consente di posizionare in modo preciso le scritte sullo schermo.*

# Eventi di mouse

Ogni componente grafico è in grado di notificare gli eventi generati dal mouse. In particolare, è predisposto per lavorare con due tipi di ascoltatori: MouseListener e MouseMotionListener. Il primo è specializzato nella gestione degli eventi relativi alla pressione dei pulsanti del mouse, mentre il secondo si occupa dello spostamento del puntatore. I metodi dell'interfaccia MouseListener intercettano la pressione di un pulsante del mouse, il suo rilascio, il clic (generato dopo una sequenza di pressione-rilascio), l'entrata e l'uscita del puntatore dall'area del componente:

**Figura 16.5 –** *Diagramma di classe degli eventi del mouse.*

```
public void mousePressed(MouseEvent e);
public void mouseReleased(MouseEvent e);
public void mouseClicked(MouseEvent e);
public void mouseEntered(MouseEvent e);
public void mouseExited(MouseEvent e);
```

MouseMotionListener, invece, ascolta lo spostamento del mouse, sia libero (mouseMoved()) sia associato alla pressione di un pulsante (mouseDragged()):

```
public void mouseMoved(MouseEvent e)
public void mouseDragged(MouseEvent e)
```

Entrambi gli ascoltatori utilizzano come evento MouseEvent, il quale dispone di un insieme di metodi che consentono di conoscere il componente che ha generato l'evento, la posizione del mouse e il numero di clic consecutivi registrati:

```
Component getComponent()
int getClickCount()
Point getPoint()
int getX()
int getY()
```

MouseEvent dispone anche di un metodo boolean isPopupTrigger(), che restituisce true quando l'evento generato può essere interpretato come una richiesta di menu contestuale secondo le convenzioni della piattaforma sottostante. L'uso di questo metodo è stato illustrato nel capitolo 13, nel paragrafo su JPopupMenu. Il package java.awt.event contiene le classi MouseAdapter e MouseMotionAdapter, che forniscono un'implementazione vuota dei due ascoltatori. Il package javax.swing.event, invece, contiene la classe MouseInputAdapter, che fornisce un'implementazione vuota di entrambe le interfacce.

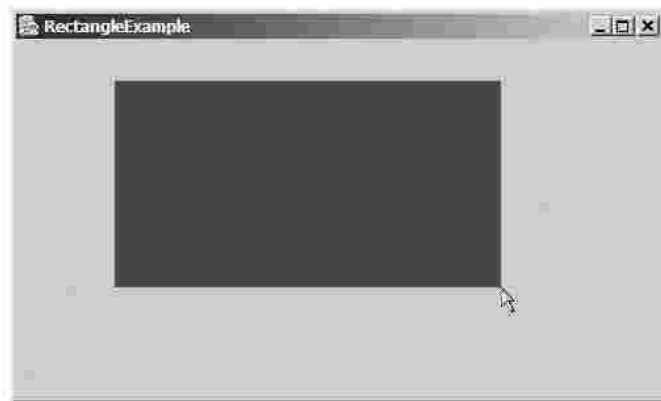
MouseEvent è una sottoclasse di InputEvent, una classe che dispone di un gruppo di metodi che permettono di sapere quali pulsanti erano premuti al momento del clic:

```
int getModifiers()
boolean isAltDown()
boolean isAltGraphDown()
boolean isControlDown()
boolean isMetaDown()
boolean isShiftDown()
```

Il prossimo esempio mostra come utilizzare gli eventi del mouse per creare semplici disegni sullo schermo. La classe principale è un JComponent, il cui metodo paintComponent() dipinge un rettangolo. Le coordinate e le dimensioni del rettangolo vengono aggiornate secondo i clic e gli spostamenti del mouse da un apposito MouseInputAdapter:

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.event.*;

public class RectangleExample extends JComponent  {
  private Point corner1 = new Point(0,0);
  private Point corner2 = new Point(0,0);

  public RectangleExample() {
    MouseInputAdapter m = new RectangleExampleMouseListener();
    addMouseListener(m);
    addMouseMotionListener(m);
```

```
    }
    public void paintComponent(Graphics g) {
      int x = Math.min(corner1.x,corner2.x);
      int y = Math.min(corner1.y,corner2.y);
      int width  = Math.abs(corner1.x - corner2.x);
      int height = Math.abs(corner1.y - corner2.y);

      g.fillRect(x, y, width, height);
    }
    class RectangleExampleMouseListener extends MouseInputAdapter {
      public void mousePressed(MouseEvent e) {
        corner1 = e.getPoint();
        corner2 = corner1;
        repaint();
      }
      public void mouseDragged(MouseEvent e) {
        corner2 = e.getPoint();
        repaint();
      }
    }
    public static void main(String argv[]) {
      JFrame f = new JFrame();
      f.setSize(500, 300);
      RectangleExample r = new RectangleExample();
      r.setForeground(Color.BLUE);
      f.getContentPane().add(r);
      f.setVisible(true);
    }
  }
```

**Figura 16.6 –** *Un componente attivo, capace di reagire agli eventi del mouse.*

## Eventi di tastiera

I componenti prevedono anche il supporto agli eventi di tastiera, mediante l'ascoltatore KeyListener, caratterizzato dai seguenti metodi:

```
void keyPressed(KeyEvent e)
void keyReleased(KeyEvent e)
void keyTyped(KeyEvent e)
```

L'oggetto KeyEvent possiede i seguenti metodi:

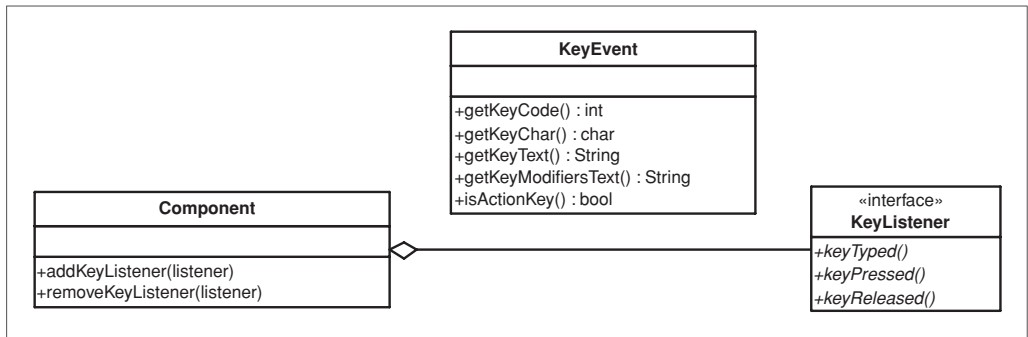
```
char getKeyChar()
int getKeyCode()
```

Il primo restituisce il carattere corrispondente al tasto premuto. Alcuni tasti non corrispondono a caratteri (Invio, Esc e così via), quindi per identificare i tasti viene usato un codice numerico detto key code. La classe KeyEvent possiede una serie di costanti corrispondenti ai codici carattere di qualsiasi tasto della tastiera. Ecco qualche esempio:

```
VK_ENTER
VK_ESCAPE
VK_EURO_SIGN
VK_F1
VK_F2
```

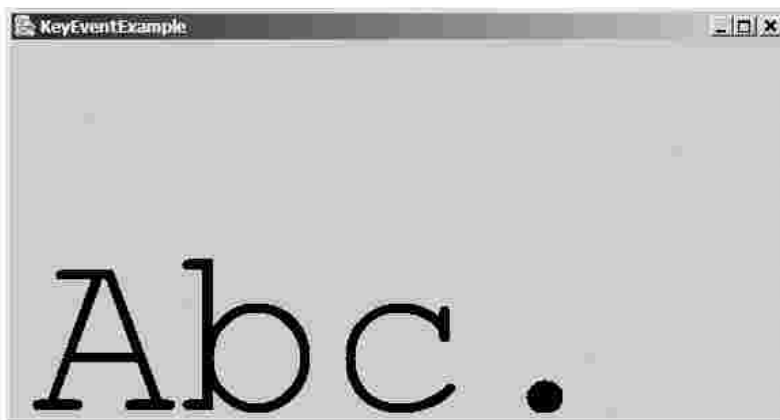
I tasti alfanumerici restituiscono anche il carattere a cui corrispondono, tramite il metodo getKeyChar(). Anche KeyEvent è sottoclasse di InputEvent e dispone dei metodi necessari a verificare la precisa combinazione dei tasti premuta dall'utente:

```
int getModifiers()
boolean isAltDown()
boolean isAltGraphDown()
boolean isControlDown()
boolean isMetaDown()
boolean isShiftDown()
```

**Figura 16.7 –** *Le classi relative alla gestione degli eventi di tastiera.*

Il seguente esempio crea un componente al cui interno viene visualizzata la lettera “A”. Premendo i tasti alfanumerici o il tasto backspace è possibile creare una frase. Mediante i tasti cursore è possibile spostare la scritta sullo schermo:

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class KeyEventExample extends JFrame {

  private int x = 30;
  private int y = 220;
  private String s = “A”;

  public KeyEventExample() {
    getContentPane().add(new InnerComponent());
    setSize(500,400);
    addKeyListener(new KeyHandler());
  }
  class InnerComponent extends JComponent {
    public void paintComponent(Graphics g) {
      g.setFont(new Font(“monospaced”,5,180));
      if(!s.equals(“”))
        g.drawString(s,x,y);
    }
  }
  class KeyHandler extends KeyAdapter {
    public void keyPressed(KeyEvent e) {
      int c = e.getKeyCode();
        switch(c) {
          case KeyEvent.VK_UP :
```

```
          y--;
          break;
        case KeyEvent.VK_DOWN :
          y++;
          break;
        case KeyEvent.VK_LEFT :
          x--;
          break;
        case KeyEvent.VK_RIGHT :
          x++;
          break;
        case KeyEvent.VK_BACK_SPACE :
          int length = s.length() > 1 ? s.length()-1 : 0;
          s = s.substring(0,length);
          break;
        default :
          s = s+Character.toString(e.getKeyChar());
          break;
      }
      repaint();
    }
  }

  public static void main(String argv[]) {
    KeyEventExample k = new KeyEventExample();
    k.setVisible(true);
  }
}
```

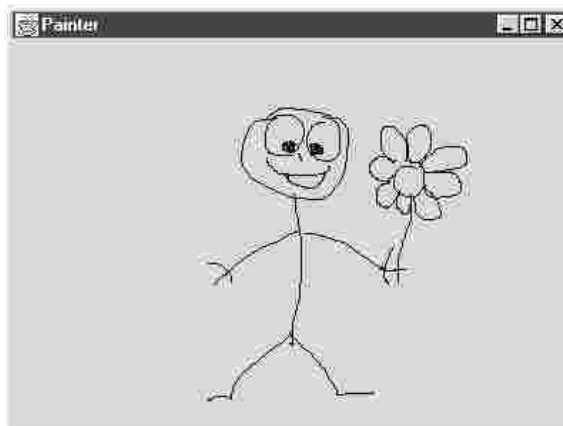
**Figura 16.8 –** *Un componente capace di reagire agli eventi della tastiera.*

## Disegno a mano libera

Per concludere questo capitolo è utile un esempio riepilogativo. Anche se i concetti appena illustrati costituiscono solo una frazione delle reali possibilità offerte da Java, essi permettono tuttavia di creare programmi grafici di una certa complessità.

Un uso adeguato degli eventi del mouse e delle primitive di disegno permette di realizzare un semplice programma di disegno a mano libera. Il disegno viene memorizzato sotto forma di poly linee: oggetti grafici composti da una sequenza di linee collegate tra loro in modo tale da dare l'illusione di un tratto continuo. Ogni volta che l'utente preme il pulsante viene creato un Vector, all'interno del quale vengono immagazzinate le coordinate di ogni punto in cui il mouse si viene a trovare durante il drag. Quando il pulsante viene rilasciato, la poly linea viene interrotta. Il Vector polyLineList contiene tutte le poly linee che compongono il disegno. Ogni volta che l'area del componente necessita di un repaint, le poly linee vengono ridipinte una a una sullo schermo:

```
import java.util.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.event.*;

public class Painter extends JComponent {

  // contiene un Vector per ogni poly line
  private Vector polyLineList = new Vector();

  // Costruttore della classe principale
  public Painter() {
    super();
    MouseInputListener m = new MyMouseInputListener();
    addMouseListener(m);
    addMouseMotionListener(m);
  }
  public void paintComponent(Graphics g) {
    // disegna ogni poly line
    Iterator polyLineIterator = polyLineList.iterator();
    while(polyLineIterator.hasNext()) {
      Vector polyLine = (Vector)polyLineIterator.next();
      Iterator pointIterator = polyLine.iterator();
      // disegna ogni linea della poly line
      Point oldPoint = (Point)pointIterator.next();
      while(pointIterator.hasNext()) {
        Point newPoint = (Point)pointIterator.next();
        g.drawLine(oldPoint.x,oldPoint.y,newPoint.x,newPoint.y);
        oldPoint = newPoint;
      }
```

```
    }
  }

  class MyMouseInputListener extends MouseInputAdapter {
    // pulsante premuto
    public void mousePressed(MouseEvent e) {
      // crea una nuova poly line e la inserisce nella lista
      Vector polyLine = new Vector();
      polyLine.add(new Point(e.getX(),e.getY()));
      polyLineList.add(polyLine);
    }
     public void mouseDragged(MouseEvent e) {
      // aggiunge un punto alla poly line
      Vector polyLine = (Vector)polyLineList.lastElement();
      polyLine.add(e.getPoint());
      repaint();
    }
  }

  public static void main(String argv[]) {
     Painter p = new Painter();
     JFrame f = new JFrame("Painter");
     f.getContentPane().add(p);
     f.setSize(400,300);
     f.setVisible(true);
  }
}
```

**Figura 16.9 –** *Le direttive di disegno e la manipolazione degli eventi del mouse permettono di esprimere il proprio estro creativo.*

# Networking

LORENZO BETTINI

## Introduzione

Si sente spesso affermare che Java è "il linguaggio di programmazione per Internet". Effettivamente la maggior parte del grande successo e della diffusione di Java è dovuta a questo, vista soprattutto l'importanza sempre maggiore che Internet sta assumendo. Java è quindi particolarmente adatto per sviluppare applicazioni che devono fare uso della rete. Ciò non deve indurre a pensare che con Java si scrivono principalmente solo Applet, per animare e rendere più carine e interattive le pagine web. Con Java si possono sviluppare vere e proprie applicazioni che devono girare in rete interagendo con più computer (le cosiddette *applicazioni distribuite*).

Non si dimentichi che un altro fattore determinante per il suo successo è l'indipendenza dalla piattaforma, ottenuta grazie all'utilizzo del bytecode. Il linguaggio astrae da problemi di portabilità come il byte ordering, e quindi anche il programmatore non deve preoccuparsi dei classici problemi di interoperabilità cross-platform.

A questo punto il programmatore di una applicazione network based non deve preoccuparsi di scrivere ex novo particolari librerie o funzioni per le operazioni di base, ma può dedicarsi totalmente ai dettegli veri e propri dell'applicazione.

Inoltre ciò che rende Java un linguaggio adatto per il networking sono le classi definite nel pacchetto java.net che sarà analizzato in questo capitolo, in cui, oltre alla descrizione delle varie classi e dei rispettivi metodi, saranno forniti anche semplici esempi estendibili e funzionanti.

# Socket

Le classi di networking incapsulano il paradigma *socket* presentato per la prima volta nella Berkeley Software Distribution (BSD) della University of California at Berkeley.

Una socket è come una porta di comunicazione e non è molto diversa da una presa elettrica: tutto ciò che è in grado di comunicare tramite il protocollo standard TCP/IP può collegarsi ad una socket e comunicare tramite questa porta, allo stesso modo in cui un qualsiasi apparecchio che funziona a corrente può collegarsi a una presa elettrica e sfruttare la tensione messa a disposizione. Nella "rete" gestita dalle socket, invece dell'elettricità, viaggiano pacchetti TCP/IP. Tale protocollo e le socket forniscono quindi un'astrazione che permette di far comunicare dispositivi diversi che utilizzano lo stesso protocollo.

Quando si parla di networking, ci si imbatte spesso nel termine *client-server*. Si tratta in realtà di un paradigma: un'entità (spesso un programma) *client* per portare a termine un particolare compito richiede dei servizi ad un'altra entità (anche questa spesso è un programma): un *server* che ha a disposizione delle risorse da condividere. Una tale situazione si ritrova spesso nell'utilizzo quotidiano dei computer (anche senza saperlo): un programma che vuole stampare qualcosa (client) richiede alla stampante (server) l'utilizzo di tale risorsa.

Il server è una risorsa costantemente disponibile, mentre il client è libero di scollegarsi dopo che è stato servito. Tramite le socket inoltre un server è in grado di servire più client contemporaneamente.

Alcuni esempi di client-server molto noti sono:

*Telnet* : se sulla nostra macchina si ha disposizione il programma Telnet (programma client), è possibile operare su un computer remoto come si opera su un computer locale. Questo è possibile se sulla macchina remota è presente un programma server in grado di esaudire le richieste del client Telnet;

*FTP* : tramite un client FTP si possono copiare e cancellare files su un computer remoto, purché qui sia presente un server FTP;

*Web* : il browser è un client web, che richiede pagine web ai vari computer su cui è installato un web server, che esaudirà le richieste spedendo la pagina desiderata.

Come si è detto, tipicamente, sia il server che il client sono dei programmi che possono girare su macchine diverse collegate in rete. Il client deve conoscere l'indirizzo del server e il particolare protocollo di comunicazione utilizzato dal server. L'indirizzo in questione è un classico *indirizzo IP*.

Un client, quindi, per comunicare con un server usando il protocollo TCP/IP dovrà per prima cosa creare una socket con tale server, specificando l'indirizzo IP della macchina su cui il server è in esecuzione e il numero di porta sulla quale il server è in ascolto. Il concetto di *porta* permette ad un singolo computer di servire più client contemporaneamente: su uno stesso computer possono essere in esecuzione server diversi, in ascolto su porte diverse. Se si vuole un'analogia si può pensare al fatto che più persone abitano nella medesima via, ma a numeri civici diversi. In questo caso i numeri civici rappresenterebbero le porte.

Un server "rimarrà in ascolto" su una determinata porta finché un client non creerà una socket con la macchina del server, specificando quella porta. Una volta che il collegamento con il server, tramite la socket è avvenuto, il client può iniziare a comunicare con il server, sfruttando la socket creata. A collegamento avvenuto si instaura un protocollo di livello superiore che dipende da quel particolare server: il client deve utilizzare quel protocollo di comunicazione, per richiedere servizi al server.

Il numero di porta è un intero compreso fra 1 e 65535. Il TCP/IP riserva le porte minori di 1024 a servizi standard. Ad esempio la porta 21 è riservata all'FTP, la 23 al Telnet, la 25 alla posta elettronica, la 80 all'HTTP (il protocollo delle pagine web), la 119 ai news server. Si deve tenere a mente che una porta in questo contesto non ha niente a che vedere con le porte di una macchina (porte seriali, parallele, ecc.), ma è un'astrazione utile per smistare informazioni a più server in esecuzione su una stessa macchina.

Si presentano adesso le classi messe a disposizione da Java nel pacchetto `java.net` per la gestione di comunicazioni in rete.

# La classe InetAddress

Come si sa, un indirizzo Internet è costituito da 4 numeri (da 0 a 255) separati ciascuno da un punto. Spesso però, quando si deve accedere a un particolare host, invece di specificare dei numeri, si utilizza un nome, che corrisponde a tale indirizzo (p.e.: `www.myhost.it`). La traduzione dal nome all'indirizzo numerico vero e proprio è compito del servizio *Domain Name Service*, abbreviato con DNS.

Senza entrare nei dettagli di questo servizio, basti sapere che la classe `InetAddress` mette a disposizione diversi metodi per astrarre dal particolare tipo di indirizzo specificato (a numeri o a lettere), occupandosi essa stessa di effettuare le dovute traduzioni.

Inoltre c'è un ulteriore vantaggio: la scelta di utilizzare un indirizzo numerico a 32 bit non fu a suo tempo una scelta molto lungimirante; con l'immensa diffusione che Internet ha avuto e sta avendo, si è molto vicini ad esaurire tutti i possibili indirizzi che si possono ottenere con 32 bit (oltretutto diversi indirizzi sono riservati e quindi il numero di indirizzi possibili si riduce ulteriormente); si stanno pertanto introducendo degli indirizzi a 128 bit che, da questo punto di vista, non dovrebbero più dare tali preoccupazioni.

Le applicazioni che utilizzeranno indirizzi Internet tramite la classe `InetAddress` saranno portabili dal punto di vista degli indirizzi, in modo completamente trasparente.

## Descrizione classe

```
public final class InetAddress extends Object implements Serializable
```

## Costruttori

La classe non mette a disposizione nessun costruttore: l'unico modo per creare un oggetto `InetAddress` prevede l'utilizzo di metodi statici, descritti di seguito.

# Metodi

```
public static InetAddress getByName(String host) throws UnknownHostException
```

Restituisce un oggetto InetAddress rappresentante l'host specificato nel parametro host. L'host può essere specificato sia come nome, che come indirizzo numerico. Se si specifica null come parametro, ci si riferisce all'indirizzo di default della macchina locale.

```
public static InetAddress[] getAllByName(String host) throws UnknownHostException
```

Tale metodo è simile al precedente, ma restituisce un array di oggetti InetAddress: spesso alcuni siti web molto trafficati registrano lo stesso nome con indirizzi IP diversi. Con questo metodo si otterranno tanti InetAddress quanti sono questi indirizzi registrati.

```
public static InetAddress getLocalHost() throws UnknownHostException
```

Viene restituito un InetAddress corrispondente alla macchina locale. Se tale macchina non è registrata, oppure è protetta da un firewall, l'indirizzo è quello di loopback: 127.0.0.1.

Tutti questi metodi possono sollevare l'eccezione UnknownHostException se l'indirizzo specificato non può essere risolto (tramite il DNS).

```
public String getHostName()
```

Restituisce il nome dell'host che corrisponde all'indirizzo IP dell'InetAddress. Se il nome non è ancora noto (ad esempio se l'oggetto è stato creato specificando un indirizzo IP numerico), verrà cercato tramite il DNS; se tale ricerca fallisce, verrà restituito l'indirizzo IP numerico (sempre sotto forma di stringa).

```
public String getHostAddress()
```

Simile al precedente: restituisce però l'indirizzo IP numerico, sotto forma di stringa, corrispondente all'oggetto InetAddress.

```
public byte[] getAddress()
```

L'indirizzo IP numerico restituito sarà sotto forma di array byte. L'ordinamento dei byte è *high byte first* (che è proprio l'ordinamento tipico della rete).

Un'Applet potrà costruire un oggetto InetAddress solo per l'host dove si trova il web server dal quale l'Applet è stata scaricata, altrimenti verrà generata un'eccezione: SecurityException.

# Un esempio

Con tale classe a disposizione è molto semplice scrivere un programma in grado di tradurre nomi di host nei corrispettivi indirizzi numerici e viceversa. Al programma che segue basterà

passare una stringa contenente o un nome di host o un indirizzo IP numerico e si avranno in risposta le varie informazioni.

```
import java.net.*;
import java.io.*;

public class HostLookup {
 public static void main(String args[]) {
   // prima si stampano i dati relativi
   // alla macchina locale...
   try {
     InetAddress LocalAddress = InetAddress.getLocalHost();
     System.out.println("host locale : "
                    + LocalAddress.getHostName() + ", IP : "
                    + LocalAddress.getHostAddress());
   } catch(UnknownHostException e) {
     System.err.println("host locale sconosciuto!");
     e.printStackTrace();
   }

   // ...poi quelli dell'host specificato
   if(args.length != 1) {
     System.err.println("Uso: HostLookup host");
   } else {
     try {
       System.out.println("Ricerca di " + args[0] + "...");
       InetAddress RemoteMachine = InetAddress.getByName(args[0]);
       System.out.println("Host Remoto : "
+ RemoteMachine.getHostName() + ", IP : "
+ RemoteMachine.getHostAddress() );
     } catch(UnknownHostException ex) {
       System.out.println("Ricerca Fallita " + args[0]);
     }
   }
 }
}
```

# URL

Tramite un URL (*Uniform Resource Locator*) è possibile riferirsi alle risorse di Internet in modo semplice e uniforme. Si ha così a disposizione una forma intelligente e pratica per identificare o indirizzare in modo univoco le informazioni su Internet. I browser utilizzano gli URL per recuperare le pagine web. Java mette a disposizione alcune classi per utilizzare gli URL; sarà così possibile, ad esempio, inglobare nelle proprie applicazioni funzioni tipiche dei web browser.

Esempi tipici di URL sono:

```
http://www.myweb.com:8080/webdir/webfile.html
ftp://ftp.myftpsite.edu/pub/programming/tips.tgz
```

Un URL consiste di 4 componenti:

1. il protocollo separato dal resto dai due punti (esempi tipici di protocolli sono http, ftp, news, file, ecc.);

2. il nome dell'host, o l'indirizzo IP dell'host, che è delimitato sulla sinistra da due barre (//), e sulla destra da una sola (/), oppure da due punti (:);

3. il numero di porta, separato dal nome dell'host sulla sinistra dai due punti, e sulla destra da una singola barra. Tale componente è opzionale, in quanto, come già detto, ogni protocollo ha una porta di default;

4. il percorso effettivo della risorsa che richiediamo. Il percorso viene specificato come si specifica un path sotto Unix. Se non viene specificato nessun file, la maggior parte dei server HTTP aggiunge automaticamente come file di default index.html.

## Descrizione classe

```
public final class URL extends Object implements Serializable
```

## Costruttori

La classe ha molti costruttori, poiché vengono considerati vari modi di specificare un URL.

```
public URL(String spec) throws MalformedURLException
```

L'URL viene specificato tramite una stringa, come ad esempio:

```
http://www.myweb.it:80/foo.html
```

```
public URL(URL context, String spec) throws MalformedURLException
```

L'URL viene creato combinando un URL già esistente e un URL specificato tramite una stringa. Se la stringa è effettivamente un URL assoluto, allora l'URL creato corrisponderà a tale stringa; altrimenti l'URL risultante sarà il percorso specificato in spec, relativo all'URL context. Ad esempio se context è http://www.myserver.it/ e spec è path/index.html, l'URL risultante sarà http://www.myserver.it/path/index.html.

```
public URL(String protocol, String host, int port,String file) throws MalformedURLException
```

Con questo costruttore si ha la possibilità di specificare separatamente ogni singolo componente di un URL.

```
public URL(String protocol, String host, String file) throws MalformedURLException
```

Simile al precedente, ma viene usata la porta di default del protocollo specificato.

Un'eccezione MalformedURLException viene lanciata se l'URL non è specificato in modo corretto (per quanto riguarda i vari componenti).

## Metodi

Di questa classe fanno parte diversi metodi che permettono di ricavare le varie parti di un URL.

```
public int getPort()
public String getProtocol()
public String getHost()
public String getFile()
```

Il significato di questi metodi dovrebbe essere chiaro: restituiscono un singolo componente dell'oggetto URL.

```
public String toExternalForm()
```

Restituisce una stringa che rappresenta l'URL

```
public URLConnection openConnection() throws IOException
```

Restituisce un oggetto URLConnection (sarà trattato di seguito), che rappresenta una connessione con l'host dell'URL, secondo il protocollo adeguato. Tramite questo oggetto, si può accedere ai contenuti dell'URL.

```
public final InputStream openStream() throws IOException
```

Apre una connessione con l'URL, e restituisce un input stream. Tale stream può essere utilizzato per leggere i contenuti dell'URL.

```
public final Object getContent() throws IOException
```

Questo metodo restituisce un oggetto di classe Object che racchiude i contenuti dell'URL. Il tipo reale dell'oggetto restituito dipende dai contenuti dell'URL: se si tratta di un'immagine, sarà un oggetto di tipo Image, se si tratta di un file di testo, sarà una String. Questo metodo compie diverse azioni, invisibili all'utente, come stabilire la connessione con il server, inviare una richiesta, processare la risposta, ecc.

## Un esempio

Ovviamente per ogni protocollo ci dovrà essere un appropriato gestore. Il JDK fornisce di default un gestore del protocollo HTTP, e quindi l'accesso alle informazioni web è alquanto semplice.

Nel caso dell'HTTP, ad esempio una chiamata al metodo `openStream`, il gestore del protocollo HTTP, invierà una richiesta al web server specificato con l'URL, analizzerà le risposte del server, e restituirà un input stream dal quale è possibile leggere i contenuti del particolare file richiesto. Richiedere un file a un server web è molto semplice, ed è illustrato nell'esempio seguente, che mostra anche l'utilizzo di altri metodi della classe.

```
import java.net.*;
import java.io.*;

public class HTTP_URL_Reader {
 public static void main(String args[]) throws IOException {
    if(args.length < 1)
       throw new IOException("Sintassi : HTTP_URL_Reader URL");

    URL url = new URL(args[0]);

    System.out.println("Componenti dell'URL");
    System.out.println("URL    : " + url.toExternalForm());
    System.out.println("Protocollo : " + url.getProtocol());
    System.out.println("Host       : " + url.getHost());
    System.out.println("Porta      : " + url.getPort());
    System.out.println("File     : " + url.getFile());

    System.out.println("Contenuto dell'URL :");

    // lettura dei dati dell'URL
    InputStream iStream = url.openStream();
    DataInputStream diStream = new DataInputStream(iStream);



String line ;
    while((line = diStream.readLine()) != null)
       System.out.println(line);

    diStream.close();
 }
}
```

È sufficiente creare un URL, passando al costruttore l'URL sotto forma di stringa, ottenere l'input stream chiamando l'apposito metodo, creare un `DataInputStream` basandosi su tale stream, e leggere una riga alla volta, stampandola sullo schermo. Il programma può essere eseguito così:

```
java HTTP_URL_Reader http://localhost/mydir/myfile.html
```

Se è installato un web server, si avranno stampate a schermo le varie componenti dell'URL specificato, nonché il contenuto del file richiesto.

# La classe URLConnection

Questa classe rappresenta una connessione attiva, specifica di un dato protocollo, a un oggetto rappresentato da un URL. Tale classe è astratta, e quindi, per gestire uno specifico protocollo, si dovrebbe derivare da questa classe.

## Descrizione classe

```
public class URLConnection extends Object
```

## Costruttori

```
protected URLConnection(URL url)
```

Crea un oggetto di questa classe, dato un URL. Da notare che il costruttore è protetto, quindi può essere chiamato solo da una classe derivata. In effetti, come si è visto nella classe URL, un oggetto di questa classe viene ottenuto tramite la chiamata del metodo `openConnection` della classe `URL`.

## Metodi

```
public URL getURL()
```

Restituisce semplicemente l'URL su cui è stato costruito l'oggetto `URLConnection`.

```
public abstract void connect() throws IOException
```

Permette di connettersi all'URL, specificato nel costruttore. La connessione quindi non avviene con la creazione dell'oggetto, ma avviene quando viene richiamato questo metodo, oppure un metodo che necessita che la connessione sia attiva (a quel punto la richiesta della connessione viene stabilita implicitamente).

```
public Object getContent() throws IOException
```

Restituisce il contenuto dell'URL. Viene restituito un oggetto di classe `Object`, poiché il tipo dell'oggetto dipende dal particolare URL.

```
public InputStream getInputStream() throws IOException
```

Restituisce un input stream con il quale si può leggere dall'URL.

```
public OutputStream getOutputStream() throws IOException
```

In questo caso si tratta di uno stream di output, con il quale è possibile inviare dati a un URL; ciò può risultare utile se si deve compiere un'operazione di post HTTP.

Questa classe contiene inoltre molti metodi che permettono di avere informazioni dettagliate sull'URL, quali il tipo di contenuto, la sua lunghezza, la data dell'ultima modifica, ecc. Per una rassegna completa si rimanda ovviamente alla documentazione on-line ufficiale.

Esistono poi alcuni metodi statici, da utilizzare per implementare gestori di protocollo personalizzati. Il trattamento di tale argomento va però oltre lo scopo di questo manuale.

# I messaggi HTTP GET e POST

I web server permettono di ottenere informazioni come risultato di una query (interrogazione). Invece di richiedere un normale documento, si specifica nell'URL il nome di un programma (che segue l'interfaccia CGI), passandogli alcuni parametri che rappresentano la query vera e propria.

Molti sostengono che l'arrivo di Java abbia decretato la morte della programmazione CGI. In effetti tramite Java si ha più flessibilità, e i programmi vengono eseguiti dal lato client. Con la programmazione CGI invece il programma viene eseguito sul server, limitando così l'interazione con l'utente. Del resto il CGI è ancora molto usato, anche perché il web è pieno di programmi CGI già scritti e collaudati. Il presente paragrafo non vuole essere una spiegazione dettagliata della programmazione CGI (di cui non sarà data nessuna descrizione approfondita), ma vuol illustrare come dialogare con programmi CGI tramite Java.

Una query CGI è costituita quindi da un normale URL, con in coda alcuni parametri. La parte dell'URL che specifica i parametri inizia con un punto interrogativo (`?`). Ogni parametro è separato da una "e commerciale" (`&`), e i valori che si assegnano ai parametri sono specificati in questo modo: `nome = valore` (il valore è facoltativo). Un esempio di query è il seguente:

```
http://localhost/cgi-bin/mycgi.exe?nome=lorenzo&cognome=bettini&eta=29
```

In questo modo si richiama il programma CGI `mycgi.exe` e ad esso si passano i valori `lorenzo`, `bettini`, `29`, da assegnare rispettivamente ai parametri `nome`, `cognome`, `eta`.

Con la classe `URL`, presente nel pacchetto `java.net`, eseguire una tale query è semplicissimo: basta passare tale stringa al costruttore della classe.

Una query spedisce quindi dei dati al web server, inserendoli direttamente nell'URL. In questo modo però si può andare incontro a problemi dovuti alla limitatezza della lunghezza delle query: non si possono spedire grandi quantità di dati in questo modo.

Per far questo si deve utilizzare un altro messaggio HTTP: il messaggio `POST`. Infatti mentre un messaggio `GET` spedisce solo un header (intestazione) nel proprio messaggio, un messaggio `POST`, oltre che di un header, è dotato anche di un contenuto (content). Vale a dire che un messaggio `POST` è molto simile, strutturalmente, ad una risposta del server web, quando si richiede un

documento (si veda a tal proposito l'esempio per ottenere una pagina web tramite le socket, nella sezione specifica). Infatti in un messaggio POST si deve includere il campo Content-length:.

Nel caso in cui si voglia inviare un messaggio POST si deve prima di tutto creare un oggetto URL, specificando un URL valido, creare un URLConnection con l'apposito metodo di URL, e abilitare la possibilità di utilizzare tale oggetto sia per l'output che per l'input. Inoltre è bene disabilitare la cache, in modo da essere sicuri che la risposta arrivi realmente dal server e non dalla cache.

```
URL destURL = new URL("http://localhost/cgi-bin/test-cgi");

URLConnection urlConn = destURL.openConnection();

urlConn.setDoOutput(true);
urlConn.setDoInput(true);
urlConn.setUseCaches(false);
```

Si deve poi riempire l'header del messaggio con alcune informazioni vitali, come il tipo del contenuto del messaggio e la lunghezza del messaggio, supponendo che il messaggio venga memorizzato in una stringa. Si ricordi che il contenuto deve essere sempre terminato da un \r\n.

```
String request = ...contenuto... + "\r\n";

urlConn.setRequestProperty("Content-type", "application/octet-stream");
urlConn.setRequestProperty("Content-length", "" + request.length());
```

A questo punto si può spedire il messaggio utilizzando lo stream dell'oggetto URLConnection (magari tramite un DataOutputStream). Dopo aver fatto questo ci si può mettere in attesa della risposta del server, sempre tramite lo stream (stavolta di input) di URLConnection.

```
DataOutputStream outStream
= new DataOutputStream(urlConn.getOutputStream());

outStream.writeBytes(request);
outStream.close();

DataInputStream inStream
= new DataInputStream(urlConn.getInputStream());

// lettura risposta dal server...
```

## La classe Socket

Per creare una socket con un server in esecuzione su un certo host è sufficiente creare un oggetto di classe Socket, specificando nel costruttore l'indirizzo internet dell'host, e il numero di porta. Dopo che l'oggetto Socket è stato costruito è possibile ottenere (tramite appositi metodi)

due stream (uno di input e uno di output). Tramite questi stream è possibile comunicare con l'host, e ricevere messaggi da esso. Qualsiasi metodo che prenda in ingresso un InputStream (o un OutputStream) può comunicare con l'host in rete.

Quindi, una volta creata la socket, è possibile comunicare in rete tramite l'usuale utilizzo degli stream.

## Descrizione classe

```
public class Socket extends Object
```

## Costruttori

```
public Socket(String host, int port) throws UnknownHostException, IOException
public Socket(InetAddress address, int port) throws IOException
```

Viene creato un oggetto Socket connettendosi con l'host specificato (sotto forma di stringa o di InetAddress) alla porta specificata. Se sull'host e sulla porta specificata non c'è un server in ascolto, verrà generata un'IOException (verrà specificato il messaggio connection refused).

## Metodi

```
public InetAddress getInetAddress()
```

Restituisce un oggetto InetAddress corrispondente all'indirizzo dell'host con il quale la socket è connessa.

```
public InetAddress getLocalAddress()
```

Restituisce un oggetto InetAddress corrispondente all'indirizzo locale al quale la socket è collegata.

```
public int getPort()
```

Restituisce il numero di porta dell'host remoto con il quale la socket è collegata.

```
public int getLocalPort()
```

Restituisce il numero di porta locale con la quale la socket è collegata. Quando si crea una socket, come si è già detto, ci si collega con un server su una certa macchina, che è in ascolto su una certa porta. Anche sulla macchina locale, sulla quale viene creata la socket, si userà per tale socket una determinata porta, assegnata dal sistema operativo, scegliendo il primo numero di porta non occupato. Si deve ricordare infatti che ogni connessione TCP consiste sempre di un indirizzo locale e di uno remoto, e di un numero di porta locale e un numero di porta remoto. Questo metodo può essere utile quando un programma, già collegato con un server remoto, crei esso stesso un server.

Per tale nuovo server può non essere specificato un numero di porta (a questo punto si prende il numero di porta assegnato dal sistema operativo). Con questo metodo si riesce a ottenere tale numero di porta, che potrà ad esempio essere comunicato ad altri programmi su altri host.

```
public InputStream getInputStream() throws IOException
public OutputStream getOutputStream() throws IOException
```

Tramite questi metodi si ottengono gli stream, per mezzo dei quali è possibile comunicare attraverso la connessione TCP instaurata con la creazione della socket. Tale comunicazione sarà quindi basata sull'utilizzo degli stream, impiegati di continuo nella programmazione in Java. Come si può notare vengono restituite InputStream e OutputStream, che sono classi astratte. In realtà vengono restituiti dei SocketInputStream e SocketOutputStream, ma tali classi non sono pubbliche. Quando si comunica attraverso connessioni TCP, i dati vengono suddivisi in pacchetti (pacchetti IP appunto), quindi è consigliabile non utilizzare tali stream direttamente, ma sarebbe meglio costruire stream "bufferizzati" evitando così di avere pacchetti contenenti poche informazioni (infatti quando si inizia a scrivere i primi byte su tali stream, verranno spediti dei pacchetti con pochi byte, o forse anche un solo byte!).

```
public synchronized void close() throws IOException
```

Con questo metodo viene chiusa la socket (e quindi la connessione), e tutte le risorse che erano in uso verranno rilasciate. Dati contenuti nel buffer verranno comunque spediti, prima della chiusura del socket. La chiusura di uno dei due stream associati alla socket comporterà automaticamente la chiusura della socket stessa.

Può essere lanciata un'IOException, a significare che ci sono stati dei problemi sulla connessione. Ad esempio quando uno dei due programmi che utilizza la socket chiude la connessione, l'altro programma potrà ricevere una tale eccezione.

```
public synchronized void setSoTimeout(int timeout) throws SocketException
```

Dopo la chiamata di tale metodo, una lettura dall'InputStream della socket bloccherà il processo solo per una quantità di tempo pari a timeout (specificato in millisecondi). Se tale lasso di tempo scade, il processo riceverà un'InterruptedIOException. La socket rimane comunque valida e riutilizzabile. Se come timeout viene specificato 0, l'attesa sarà illimitata (infinita), che è anche la situazione di default.

```
public synchronized int getSoTimeout() throws SocketException
```

Con questo metodo si può ottenere il timeout settato con il precedente metodo. Se viene restituito 0, vuol dire che non è stato settato nessun timeout.

Quindi connettersi e comunicare con un server è molto semplice: basta creare una socket specificando host e porta (queste informazioni devono essere conosciute), ottenere e memorizzare gli stream della socket richiamando gli appositi metodi della socket, e utilizzarli per comunicare (sia per spedire informazioni, che per ricevere informazioni), magari dopo aver "bufferizzato" tali stream.

# Utilizzo delle socket (client-server)

Si prenderà adesso in esame un semplice programma client: si tratta di un client HTTP che, dato un URL, richiede un file al server HTTP di quell'host. Si tratta di una variazione di HTTP_URL_Reader visto precedentemente durante la spiegazione della classe URL.

```
import java.net.*;
import java.io.*;

public class HTTPClient {
  public HTTPClient(String textURL) throws IOException {
    Socket socket = null;
    dissectURL(textURL);
    socket = connect();
    try {
      getPage();
    } finally {
      socket.close();
    }
  }

  protected String host, file;
  protected int port;

  protected void dissectURL(String textURL) throws MalformedURLException {
    URL url = new URL(textURL);
    host = url.getHost();
    port = url.getPort();
    if(port == -1)
      port = 80;
    file = url.getFile();
  }

  protected DataInputStream in;
  protected DataOutputStream out;

  protected Socket connect() throws IOException {
    System.err.println("Connessione a " + host + ":" + port + "...");
    Socket socket = new Socket(host, port);
    System.err.println("Connessione avvenuta.");

    BufferedOutputStream buffOut = new BufferedOutputStream(socket.getOutputStream());
    out = new DataOutputStream(buffOut);
    in = new DataInputStream(socket.getInputStream());

    return socket;
  }
```

```
protected void getPage() throws IOException {
   System.err.println("Richiesta del file " + file + " inviata...");
   out.writeBytes("GET " + file + " HTTP/1.0\r\n\r\n");
   out.flush();

   System.err.println("Ricezione dati...");

   String input ;
   while((input = in.readLine()) != null)
      System.out.println(input);
}

public static void main(String args[]) throws IOException {
   if(args.length < 1)
      throw new IOException("Sintassi : HTTPClient URL");

   try {
      new HTTPClient(args[0]);
   } catch(IOException ex) {
      ex.printStackTrace();
   }

   System.out.println("exit");
}
}
```

In effetti è stata ancora utilizzata questa classe per gestire l'URL passato sulla linea di comando, ma poi si effettua una connessione con il server creando esplicitamente una socket.

Nel metodo connect si effettua la connessione vera e propria aprendo una socket con l'host e sulla porta specificati:

```
Socket socket = new Socket(host, port);
```

Effettuata la connessione si possono ottenere gli stream associati con i metodi getOutputStream e getInputStream della classe Socket. Si crea poi un DataOutputStream e un DataInputStream su tali stream ottenuti (effettivamente, per ottimizzare la comunicazione in rete, prima viene creato uno stream "bufferizzato" sullo stream di output, ma questi dettagli, al momento, possono non essere approfonditi).

A questo punto si deve richiedere il file al server web e quindi si spedisce tale richiesta tramite lo stream di output:

```
out.writeBytes("GET " + file + " HTTP/1.0\r\n\r\n");
```

Ora non resta che mettersi in attesa, sullo stream di input, dell'invio dei dati dal server:

```
while((input = in.readLine()) != null)
```

Il contenuto del file viene stampato sullo schermo una riga alla volta.

Questo semplice programma illustra un esempio di client che invia al server una richiesta, e riceve dal server i dati richiesti. Questo è quanto avviene quando, tramite il proprio browser, si visita una pagina web: anche se in modo senz'altro più complesso il client apre una connessione con il server, comunica al server quello che desidera tramite un protocollo di comunicazione (nell'esempio, l'HTTP), e attende la risposta del server (comunicata sempre tramite lo stesso protocollo). Il comando GET infatti fa parte del protocollo HTTP.

Per testare il programma non è necessaria una connessione a Internet, basta avere un web server installato e attivo e lanciare il programma in questo modo:

```
java HTTPClient http://localhost/index.html
```

E sullo schermo verrà stampato il contenuto dell'intero file index.html (se il file viene trovato, ovviamente, altrimenti si otterrà il tipico errore di file non trovato a cui ormai la navigazione web ci ha abituati).

Si vedrà adesso un esempio di programma *server*. Un server rimane in attesa di connessioni su una certa porta e, ogni volta che un client si connette a tale porta, il server ottiene una socket, tramite la quale può comunicare con il client. Il meccanismo messo a disposizione da Java per queste operazioni è la classe ServerSocket, tramite la quale il server può appunto accettare connessioni dai client attraverso la rete.

---

I passi tipici di un server saranno quindi:

1. creare un oggetto di classe ServerSocket specificando un numero di porta locale;

2. attendere (tramite il metodo accept() di suddetta classe) connessioni dai client;

3. usare la socket ottenuta ad ogni connessione, per comunicare con il client.

Infatti il metodo accept() della classe ServerSocket crea un oggetto Socket per ogni connessione. Il server potrà poi comunicare come fa un client: estraendo gli stream di input ed output dalla socket.

---

Tali passi possono essere riassunti nel seguente estratto di listato:

```
ServerSocket server = new ServerSocket(port);
Socket client = server.accept();
server.close();

InputStream i = client.getInputStream();
OutputStream o = client.getOutputStream();
```

Il server dell'esempio precedente chiude il ServerSocket appena ha ricevuto una richiesta di connessione, quindi tale server funziona una sola volta! Si ricorda che tale chiusura non chiude la connessione con il client appena creata: semplicemente il server non accetta ulteriori connessioni. Un server che "si rispetti", invece deve essere in grado di accettare più connessioni e, inoltre, dovrebbe essere in grado di soddisfare più richieste contemporaneamente. Per risolvere questo problema si deve ricorrere al *multithreading*, per il quale Java offre diversi strumenti. Il programma sarà modificato nei due punti seguenti: il thread principale rimarrà in ascolto di richieste di connessioni; appena arriva una richiesta di connessione viene creato un thread che si occuperà di tale connessione e il thread principale tornerà ad aspettare nuove connessioni.

In effetti è questo quello che avviene nei server di cui si è già parlato. Se si osserva un programma scritto in C che utilizza le socket, si potrà vedere che appena viene ricevuta una richiesta di connessione, il programma si duplica (esegue una fork()), e il programma figlio lancia un programma che si occuperà di gestire la connessione appena ricevuta. Nel caso in esame basterà creare un nuovo thread e passargli la socket della nuova connessione.

Segue il programma modificato per trattare più connessioni contemporaneamente:

```
import java.net.*;
import java.io.*;

public class SimpleServer extends Thread {
 protected Socket client ;

 public SimpleServer(Socket socket) {
   System.out.println("Arrivato un nuovo client da " + socket.getInetAddress()) ;
   client = socket;
 }

 public void run() {
   try {
     InputStream i = client.getInputStream();
     OutputStream o = client.getOutputStream();
     PrintStream p = new PrintStream(o);
     p.println("BENVENUTI.");
     p.println("Questo è il SimpleServer :-)");
     p.println();
     p.println("digitare HELP per la lista di servizi disponibili");

     int x;
     ByteArrayOutputStream command =new ByteArrayOutputStream();
     String HelpCommand = new String("HELP");
     String QuitCommand = new String("QUIT");
     while((x = i.read()) > -1) {
       o.write(x);
       if(x == 13) { /* newline */
         p.println() ;
```

```
            if(HelpCommand.equalsIgnoreCase(command.toString())) {
               p.println("Il solo servizio disponibile è l'help,");
               p.println("e QUIT per uscire.");
               p.println("Altrimenti che SimpleServer sarebbe... ;-)");
            } else if(QuitCommand.equalsIgnoreCase(command.toString())) {
               p.println("Grazie per aver usato SimpleServer ;-)");
               p.println("Alla prossima. BYE");
               try {
                  Thread.sleep(1000);
               } finally {
                  break;

            }
            } else {
               p.println("Comando non disponibile |-( ");
               p.println("Digitare HELP per la lista dei servizi");
            }
            command.reset();
         } else if( x != 10 ) /* carriage return */
            command.write(x);
      }
   } catch(IOException e) {
         e.printStackTrace();
   } finally {
      System.out.println("Connessione chiusa con " + client.getInetAddress());
      try {
         client.close();
      } catch(IOException e) {
         e.printStackTrace();
      }
   }
}

public static void main(String args[]) throws IOException {
   int port = 0;
   Socket client;

   if(args.length == 1)
      port = Integer.parseInt(args[0]) ;

   System.out.println("Server in partenza sulla porta " + port);
   ServerSocket server = new ServerSocket(port);
   System.out.println("Server partito sulla porta " + server.getLocalPort() );

   while(true) {
      System.out.println("In attesa di connessioni...");
      client = server.accept();
```

```
            System.out.println("Richiesta di connessione da " + client.getInetAddress());
            (new SimpleServer(client)).start();
        }
    }
}
```

Questo programma accetta da linea di comando un parametro che specifica la porta su cui mettersi in ascolto di richieste di connessioni. Se non viene passato alcun argomento si userà la porta scelta dal sistema operativo. Dopo la creazione dell'oggetto ServerSocket ci si mette in ascolto di connessioni e, appena se ne riceve una, si fa partire un Thread per gestire tale connessione. In effetti tale classe deriva dalla classe Thread, e quindi, quando si crea un oggetto di questa classe, si crea in effetti un nuovo thread di esecuzione. In pratica si può riassumere:

- nel main si entra in un ciclo infinito (il ciclo finirà quando viene sollevata un'eccezione, oppure quando interrompiamo il programma), in cui viene eseguito accept();

- appena viene ricevuta una richiesta di connessione si crea un nuovo oggetto della classe (e quindi un nuovo thread), passando ad esso la socket relativa a tale connessione, e viene lanciato così un nuovo thread di esecuzione;

- si torna ad eseguire l'accept();

- il codice che si occupa della comunicazione con il client è nel metodo run che viene chiamato automaticamente quando un thread viene mandato in esecuzione (cioè quando si richiama il metodo start()).

Tramite l'oggetto Socket restituito dal metodo accept si ottengono i due stream per comunicare con il client. Si attende poi che il client invii dei comandi: ogni volta che viene letto un carattere, questo viene rispedito al client, in modo che quest'ultimo possa vedere quello che sta inviando. Appena viene digitato un newline (cioè invio o enter) si controlla se il servizio richiesto (memorizzato via via in un buffer) è disponibile, e si risponde in modo opportuno. Si noti come tutte le comunicazioni fra il server e il client siano racchiuse in un blocco try–finally: se nel frattempo avviene un'eccezione, si è comunque sicuri che la connessione verrà chiusa. L'eccezione in questione è tipicamente una IOException dovuta alla disconnessione da parte del client.

La classe deriva dalla classe Thread. È da notare come — poiché il metodo run della classe Thread, che viene ridefinito dalla nostra classe, non lancia nessuna eccezione — dobbiamo intercettare tutte le eccezioni all'interno del metodo: in questo caso l'eccezione in questione è IOException che può essere lanciata anche quando si cerca di chiudere la comunicazione. A proposito di client: in questo esempio, dov'è il client? Come nell'altro esempio avevamo usato un server già esistente (web server) per testare il nostro client, questa volta per testare il nostro server utilizzeremo un client classico: il *Telnet*.

Quindi se si è lanciato il server con la seguente riga di comando

```
java SimpleServer 9999
```

basterà utilizzare da un'altro terminale (ad esempio un'altra shell del DOS in Windows, o un altro xterm sotto Linux) il seguente comando:

```
telnet localhost 9999
```

Adesso è possibile inviare richieste al server semplicemente inserendo una stringa e premendo INVIO (provate ad esempio con "HELP").

# User Datagram Protocol (UDP)

Finora si è sempre parlato del TCP (*Transfer Control Protocol*), un protocollo sviluppato sopra l'IP (*Internet Protocol*). Un altro protocollo basato sempre sull'IP, è l'UDP (*User Datagram Protocol*). In questo protocollo vengono spediti pacchetti di informazioni. Si tratta di un protocollo non basato sulla connessione (*connectionless*) e che non garantisce né l'arrivo né l'ordine dei pacchetti. Comunque, se i pacchetti arrivano, è garantito che siano integri e non corrotti. In un protocollo basato sulla connessione, come il TCP, si deve prima di tutto stabilire la connessione, dopo di che tale connessione può essere utilizzata sia per spedire che per ricevere. Quando la comunicazione è terminata, la connessione dovrà essere chiusa. Nell'UDP, invece, ogni messaggio sarà spedito come un pacchetto indipendente, che seguirà un percorso indipendente. Oltre a non garantire l'arrivo di tali pacchetti il protocollo non garantisce nemmeno che i pacchetti arrivino nell'ordine in cui sono stati spediti, e che non ci siano duplicati. Entrambi i protocolli utilizzano pacchetti, ma l'UDP, da questo punto di vista, è molto più vicino all'IP.

Ma perché utilizzare un protocollo così poco "affidabile"? Si tenga presente, che rispetto al TCP, l'UDP ha molto poco overhead (dovendo fare molti meno controlli), quindi può essere utilizzato quando la latenza è di fondamentale importanza. La perdita di pacchetti UDP è dovuta sostanzialmente alla congestione della rete. Utilizzando Internet questo è molto comune, ma se si utilizza una rete locale, questo non dovrebbe succedere.

Si può, comunque, aggiungere manualmente un po' di controllo sulla spedizione dei pacchetti. Si può supporre che, se non si riceve una risposta entro un certo tempo, il pacchetto sia andato perso, e quindi può essere rispedito. Va notato che, se il server ha ricevuto il pacchetto ma è la sua risposta che ha trovato traffico nella rete, il server riceverà nuovamente un pacchetto identico al precedente!

Si potrebbe allora pensare di implementare ulteriori controlli, ma questo porterebbe sempre più vicini al TCP. Nel caso in cui si voglia avere sicurezza sulla qualità di pacchetti, conviene passare direttamente al protocollo TCP appositamente pensato per questo scopo.

# La classe DatagramPacket

Si devono creare oggetti di questa classe sia per spedire, sia per ricevere pacchetti. Un pacchetto sarà costituito dal messaggio vero e proprio e dall'indirizzo di destinazione. Per ricevere un pacchetto UDP si dovrà costruire un oggetto di questa classe e accettare un pacchetto UDP dalla rete. Non si possono filtrare tali pacchetti: si ricevono tutti i pacchetti UDP con il proprio indirizzo.

## Descrizione classe

public final class DatagramPacket extends Object

## Costruttori

public DatagramPacket(byte ibuf[], int ilength, InetAddress iaddr, int iport)

Si costruisce un datagram packet specificando il contenuto del messaggio (i primi ilength bytes dell'array ibuf) e l'indirizzo IP del destinatario (sempre nella forma "host + numero porta").

> Importante: poiché si tratta di protocolli differenti, un server UDP ed uno TCP possono essere in ascolto sulla stessa porta.

public DatagramPacket(byte ibuf[], int ilength)

In questo modo si costruisce un oggetto DatagramPacket da utilizzare per ricevere pacchetti UDP dalla rete. Il pacchetto ricevuto sarà memorizzato nell'array ibuf che dovrà essere in grado di contenere il pacchetto. ilenght specifica la dimensione massima di bytes che potranno essere ricevuti.

## Metodi

Vi sono alcuni metodi che permettono di leggere le informazioni e il contenuto di un pacchetto UDP.

public InetAddress getAddress()

Se il pacchetto è stato ricevuto, tale metodo restituirà l'indirizzo dello host mittente; se l'oggetto DatagramSocket è invece stato creato per essere trasmesso, conterrà l'indirizzo IP del destinatario.

public int getPort()

Restituisce il numero del mittente o destinatario (vedi sopra), che può essere utilizzato per rispondere.

public byte[] getData()

Estrae dal pacchetto il contenuto del messaggio. L'array avrà la stesa grandezza specificata nel costruttore, e non l'effettiva dimensione del messaggio. Per ottenere tale dimensione si deve utilizzare il seguente metodo:

public int getLength()

# La classe DatagramSocket

Questa classe permette di spedire e ricevere pacchetti UDP (sempre utilizzando le socket). Quando si spedisce un pacchetto UDP, come nel TCP, ci deve essere un DatagramSocket in ascolto sulla porta specificata.

Trattandosi di un protocollo connectionless, lo stesso oggetto DatagramSocket può essere utilizzato per spedire pacchetti a host differenti e ricevere pacchetti da host diversi.

## Descrizione classe

```
public class DatagramSocket extends Object
```

## Costruttori

Si può specificare il numero di porta, oppure lasciare che sia il sistema operativo ad assegnarla. Tipicamente se si deve spedire un pacchetto (client) si utilizzerà la porta assegnata dal sistema operativo, e se si deve ricevere (server) si specificherà un numero di porta preciso. Ovviamente tale numero dovrà essere noto anche ai client. Ci sono quindi due costruttori:

```
public DatagramSocket() throws SocketException

public DatagramSocket(int port) throws SocketException
```

Si può inoltre specificare anche l'indirizzo al quale tale socket sarà legata:

```
public DatagramSocket(int port, InetAddress laddr) throws SocketException
```

## Metodi

```
public void send(DatagramPacket p) throws IOException
```

Spedisce un pacchetto all'indirizzo di destinazione.

```
public synchronized void receive(DatagramPacket p) throws IOException
```

Riceve un singolo pacchetto UDP memorizzandolo in p. A questo punto si potranno ottenere tutte le informazioni su tale pacchetto, con i metodi della classe DatagramPacket.

```
public InetAddress getLocalAddress()
public int getLocalPort()
public synchronized void setSoTimeout(int timeout) throws SocketException
public synchronized int getSoTimeout() throws SocketException
public void close()
```

Questi metodi hanno lo stesso significato degli omonimi metodi della classe Socket; si rimanda quindi alla trattazione di tale classe.

## Un esempio

Ecco un semplice esempio di utilizzo del protocollo UDP, tramite le due classi appena illustrate. Si tratta di due classi UDPSender e UDPReceiver, il cui nome dovrebbe essere esplicativo circa il loro funzionamento.

Ecco UDPSender:

```
import java.net.*;
import java.io.*;

public class UDPSender {
 static protected DatagramPacket buildPacket(String host, int port, String message) throws IOException {
    ByteArrayOutputStream boStream = new ByteArrayOutputStream();
    DataOutputStream doStream = new DataOutputStream(boStream);
    doStream.writeUTF(message);
    byte[] data = boStream.toByteArray();
    return new DatagramPacket(data, data.length,
                                     netAddress.getByName(host), port);
 }

 public static void main(String args[]) throws IOException {
    if(args.length < 3)
       throw new IOException("Uso: UDPSender <host> <port> <messaggio> {messaggi}");

    DatagramSocket dsocket = new DatagramSocket();
    DatagramPacket dpacket ;

    for(int i = 2; i < args.length; i++) {
       dpacket = buildPacket(args[0], Integer.parseInt(args[1]), args[i]);
       dsocket.send(dpacket);
       System.out.println("Messaggio spedito");
    }
 }
}
```

e UDPReceiver:

```
import java.net.*;
import java.io.*;

public class UDPReceiver {
```

```
static protected void showPacket(DatagramPacket p) throws IOException {
   System.out.println("Mittente : " + p.getAddress());
   System.out.println("porta  : " + p.getPort()) ;
   System.out.println("Lunghezza messaggio : " + p.getLength());
   ByteArrayInputStream biStream
   = new ByteArrayInputStream(p.getData(), 0, p.getLength());
   DataInputStream diStream = new DataInputStream(biStream);
   String content = diStream.readUTF();
   System.out.println("Messaggio : " + content);
}

public static void main(String args[]) throws IOException {
   if(args.length != 1)
      throw new IOException("Uso: UDPReceiver <port>");

   byte buffer[] = new byte[65536];
   DatagramSocket dsocket
= new DatagramSocket(Integer.parseInt(args[0]));
   DatagramPacket dpacket;

   while(true) {
      System.out.println("In attesa di messaggi...");
      dpacket = new DatagramPacket(buffer, buffer.length);
      dsocket.receive(dpacket);
      System.out.println("Ricevuto messaggio");
      showPacket(dpacket);
   }
}
}
```

Si dovrà lanciare prima l'UDPReceiver specificando semplicemente il numero di porta su cui rimarrà in ascolto:

```
java UDPReceiver 9999
```

E su un altro terminale si potrà lanciare il sender specificando l'indirizzo e la porta, e poi una serie di stringhe, che verranno inviate al receiver:

```
java UDPSender localhost 9999 ciao a tutti
```

A questo punto il receiver mostrerà le informazioni riguardanti ogni messaggio ricevuto.

Su altri terminali si possono lanciare altri sender, sempre diretti allo stesso receiver, e si potrà notare che il receiver potrà ricevere messaggi da più sender, tramite lo stesso DatagramSocket. Non trattandosi di un protocollo con connessione, il socket rimarrà attivo anche quando i sender termineranno, cosa che non accade quando si crea una connessione diretta tramite una socket nel TCP.

# Nuove estensioni e classi di utility presenti nella piattaforma Java 2

A partire dalla versione 2 del linguaggio sono state aggiunte al package java.net alcune classi di utilità che offrono un maggiore livello di astrazione o mettono a disposizione alcune *feature* ormai comuni nell'ambito del networking. Queste classi sono dedicate principalmente allo sviluppo di applicazioni che si appoggiano sul protocollo HTTP. Si vedranno qui di seguito, sinteticamente, alcune di queste classi con la descrizione dei principali metodi.

# La classe HttpURLConnection

Estensione della classe URLConnection, questa classe mette a disposizione alcuni metodi specifici per il protocollo HTTP, metodi che permettono di tralasciare alcuni dettagli implementativi del protocollo stesso.

Il costruttore della classe ha la seguente firma

```
protected HttpURLConnection(URL myUrl)
```

Anche in questo caso, analogamente alla URLConnection, il costruttore è protected; per ottenere un oggetto di questa classe sarà sufficiente ricorrere allo stesso sistema con cui si ottiene un URLConnection, preoccupandosi di eseguire il cast opportuno.

Ad esempio si può scrivere

```
URL url = new URL(name);
HttpURLConnection conn = (HttpURLConnection) url.openConnection();
```

## Metodi

```
public InputStream getErrorStream()
```

In caso di fallimento della connessione, permette di utilizzare lo stream restituito per ottenere le informazioni eventualmente inviate dal server sulle cause del fallimento.

```
public boolean getFollowRedirect()
```

Restituisce true se questa connessione è abilitata a seguire le redirezioni indicate dal server, false altrimenti (vedi oltre: setFollowRedirect()).

```
public Permission getPermission()
```

Restituisce un oggetto di tipo Permission, contenente i permessi necessari a eseguire questa connessione.

```
public String getRequestMethod()
```

Restituisce il metodo richiesto per questa connessione (POST, GET, ecc.) (vedi oltre: setRequestMethod()).

```
public int getResponseCode()
```

Restituisce il codice di stato della richiesta, inviato dal server.

```
public String getResponseMessage()
```

Restituisce il messaggio di risposta del server, collegato al codice.

```
public static void setFollowRedirect(boolean set)
```

Permette di configurare il comportamento di questa connessione a fronte di una richiesta di redirezione da parte del server.

```
public void setRequestMethod(String method) throws ProtocolException
```

Utilizzato per settare il metodo voluto per questa connessione. Il parametro è tipicamente una stringa indicante una delle operazioni previste dal protocollo HTTP, ad esempio GET, POST, ecc.; il metodo di default è GET.

## La classe JarURLConnection

Questa classe astrae la connessione (HTTP) verso file archivio .jar remoti: il suo utilizzo si dimostra utile ad esempio nelle Applet, per accedere a file di immagini già presenti nella cache del browser.

Il meccanismo per ottenere un oggetto di tipo JarURLConnection è analogo a quello per HttpURLConnection; da notare in questo caso che, nella creazione della URL, è necessario specificare nel protocollo che si richiede una connessione ad un file .jar.

```
URL url = new URL(“jar:http://www.my.address/jarfile.jar”);
```

Per ottenere tutto il file.

```
URL url = new URL(“jar:http://www.my.address/jarfile.jar/adirectory/afilename”);
```

Per ottenere un file contenuto all’interno dell’archivio.

```
URL url = new URL(“jar:http://www.my.address/jarfile.jar/adirectory”);
```

Per ottenere una directory contenuta nell'archivio.

Anche in questo caso per il costruttore

```
protected JarURLConnection(URL url)
```

vale quanto illustrato per la classe HttpURLConnection.

## Metodi

```
public String getEntryName()
```

Restituisce la entry name se la connessione punta a un file contenuto in un archivio, null altrimenti.

```
public JarEntry getJarEntry()
```

Con questo metodo è possibile ottenere la JarEntry riferita all'oggetto della connessione; attraverso la JarEntry è possibile ottenere informazioni sull'oggetto della connessione, quali la dimensione in bytes, il metodo di compressione, ecc.

```
public JarFile getJarFile()
```

Restituisce il file .jar a cui fa riferimento questa connessione. Da notare che il file in questione non è modificabile.

```
public Manifest getManifest()
```

Restituisce, se esiste, il file Manifest contenuto nell'archivio.

# JavaBeans

Andrea Gini

## La programmazione a componenti

Uno degli obiettivi più ambiziosi dell'ingegneria del software è organizzare lo sviluppo di sistemi in maniera simile a quanto è stato fatto in altre branche dell'ingegneria, dove la presenza di un mercato di parti standard altamente riutilizzabili permette di aumentare la produttività riducendo nel contempo i costi. Nella meccanica, ad esempio, esiste da tempo un importante mercato di componenti riutilizzabili, come viti, dadi, bulloni e ruote dentate; ciascuno di questi componenti trova facilmente posto in centinaia di prodotti diversi.

L'industria del software, sempre più orientata alla filosofia dei componenti, sta dando vita a due nuove figure di programmatore: il progettista di componenti e l'assemblatore di applicazioni.

Il primo ha il compito di scoprire e progettare oggetti software di uso comune, che possano essere utilizzati con successo in contesti differenti. Produttori in concorrenza tra di loro possono realizzare componenti compatibili, ma con caratteristiche prestazionali differenti. L'acquirente può orientarsi su un mercato che offre una pluralità di scelte e decidere in base al budget o a particolari esigenze di prestazione.

L'assemblatore di applicazioni, d'altra parte, è un professionista specializzato in un particolare dominio applicativo, capace di creare programmi complessi acquistando sul mercato componenti standard e combinandoli con strumenti grafici o linguaggi di scripting.

Questo capitolo offre un'analisi approfondita delle problematiche che si incontrano nella creazione di componenti in Java; attraverso gli esempi verrà comunque offerta una panoramica su come sia possibile assemblare applicazioni complesse a partire da componenti concepiti per il riuso.

# La specifica JavaBeans

JavaBeans è una specifica, ossia un insieme di regole seguendo le quali è possibile realizzare in Java componenti software riutilizzabili, che abbiano la capacità di interagire con altri componenti, realizzati da altri produttori, attraverso un protocollo di comunicazione comune.

Ogni Bean è caratterizzato dai servizi che è in grado di offrire e può essere utilizzato in un ambiente di sviluppo differente rispetto a quello in cui è stato realizzato. Quest'ultimo punto è cruciale nella filosofia dei componenti: sebbene i Java Beans siano a tutti gli effetti classi Java, e possano essere manipolati completamente per via programmatica, essi vengono spesso utilizzati in ambienti di sviluppo diversi, come tool grafici o linguaggi di scripting.

I tool grafici, tipo JBuilder, permettono di manipolare i componenti in maniera visuale. Un assemblatore di componenti può selezionare i Beans da una palette, inserirli in un apposito contenitore, impostarne le proprietà, collegare gli eventi di un Bean ai metodi di un altro, generando in tal modo applicazioni, Applet, Servlet e persino nuovi componenti senza scrivere una sola riga di codice.

I linguaggi di scripting, di contro, offrono una maggiore flessibilità rispetto ai tool grafici, senza presentare le complicazioni di un linguaggio generico. La programmazione di pagine web dinamiche, uno dei domini applicativi di maggior attualità, deve il suo rapido sviluppo a un'intelligente politica di stratificazione, che vede le funzionalità di più basso livello, come la gestione dei database, la Businnes Logic o l'interfacciamento con le risorse di sistema, incapsulate all'interno di JavaBeans, mentre tutto l'aspetto della presentazione viene sviluppato con un semplice linguaggio di scripting, tipo Java Server Pages o PHP.

# Il modello a componenti JavaBeans

Un modello a componenti è caratterizzato da almeno sette fattori: proprietà, metodi, introspezione, personalizzazione, persistenza, eventi e modalità di deployment. Nei prossimi paragrafi si analizzerà il ruolo di ciascuno di questi aspetti all'interno della specifica Java Beans; quindi si procederà a descriverne l'implementazione in Java.

## Proprietà

Le proprietà sono attributi privati, accessibili solamente attraverso appositi metodi `get` e `set`. Tali metodi costituiscono l'unica via di accesso pubblica alle proprietà, cosa che permette al progettista di componenti di stabilire per ogni parametro precise regole di accesso. Se si utilizzano i Bean all'interno di un programma di sviluppo visuale, le proprietà di un componente vengono visualizzate in un apposito pannello, che permette di modificarne il valore con un opportuno strumento grafico.

## Metodi

I metodi di un Bean sono metodi pubblici Java, con l'unica differenza che essi risultano accessibili anche attraverso linguaggi di scripting e Builder Tools. I metodi sono la prima e più importante via d'accesso ai servizi di un Bean.

## Introspezione

I Builder Tools scoprono i servizi di un Bean (proprietà, metodi ed eventi) attraverso un processo noto come introspezione, che consiste principalmente nell'interrogare il componente per conoscerne i metodi, e dedurre da questi le caratteristiche. Il progettista di componenti può attivare l'introspezione in due maniere: seguendo precise convenzioni nella formulazione delle firme dei metodi, o creando una speciale classe BeanInfo, che fornisce un elenco esplicito dei servizi di un particolare Bean.

La prima via è senza dubbio la più semplice: se si definiscono i metodi di accesso a un determinato servizio seguendo le regole di naming descritte dalla specifica JavaBeans, i tool grafici saranno in grado, grazie alla reflection, di individuare i servizi di un Bean semplicemente osservandone l'interfaccia di programmazione. Il ricorso ai BeanInfo, d'altro canto, torna utile in tutti quei casi in cui sia necessario mascherare alcuni metodi, in modo da esporre solamente un sottoinsieme dei servizi effettivi del Bean.

## Personalizzazione

Durante il lavoro di composizione di Java Beans all'interno di un tool grafico, un apposito Property Sheet, generato al volo dal programma di composizione, mostra lo stato delle proprietà e permette di modificarle con un opportuno strumento grafico, tipo un campo di testo per valori String o una palette per proprietà Color. Simili strumenti grafici vengono detti editor di proprietà.

I tool grafici dispongono di editor di proprietà in grado di supportare i tipi Java più comuni, come i tipi numerici, le stringhe e i colori; nel caso si desideri rendere editabile una proprietà di un tipo diverso, è necessario realizzare un'opportuna classe di supporto, conforme all'interfaccia PropertyEditor. Quando invece si desideri fornire un controllo totale sulla configurazione di un Bean, è possibile definire un Bean Customizer, una speciale applicazione grafica specializzata nella configurazione di un particolare tipo di componenti.

## Persistenza

La persistenza permette ad un Bean di salvare il proprio stato e di ripristinarlo in un secondo tempo. JavaBeans supporta la persistenza grazie all'Object Serialization, che permette di risolvere questo problema in modo molto rapido.

## Eventi

Nella programmazione a oggetti tradizionale non esiste nessuna convenzione su come modellare lo scambio di messaggi tra oggetti. Ogni programmatore adotta un proprio sistema, creando una fitta rete di dipendenze che rende molto difficile il riutilizzo di oggetti in contesti differenti da quello di partenza. Gli oggetti Java progettati secondo la specifica Java Beans adottano un meccanismo di comunicazione basato sugli eventi, simile a quello utilizzato nei componenti grafici Swing e AWT. L'esistenza di un unico protocollo di comunicazione standard garantisce l'intercomunicabilità tra componenti, indipendentemente da chi li abbia prodotti.

## Deployment

I JavaBeans possono essere consegnati, in gruppo o singolarmente, attraverso file JAR, speciali archivi compressi in grado di trasportare tutto quello di cui un Bean ha bisogno, come classi, immagini o altri file di supporto. Grazie ai file .jar è possibile consegnare i Beans con una modalità del tipo "chiavi in mano": l'acquirente deve solamente caricare un file JAR nel proprio ambiente di sviluppo e i Beans in esso contenuti verranno subito messi a disposizione. L'impacchettamento di classi Java all'interno di file JAR segue poche semplici regole, che verranno descritte negli esempi del capitolo.

# Guida all'implementazione dei JavaBeans

Realizzare un componente Java Bean è un compito alla portata di qualunque programmatore Java che disponga di buone conoscenze di sviluppo Object Oriented. Nei paragrafi seguenti verranno descritte dettagliatamente le convenzioni di naming dettate dalla specifica, e verranno fornite le istruzioni su come scrivere le poche righe di codice necessarie a implementare i meccanismi che caratterizzano i servizi Bean. Infine verranno presentati degli esempi, che permetteranno di impratichirsi con il processo di implementazione delle specifiche.

## Le proprietà

Le proprietà sono attributi che descrivono l'aspetto e il comportamento di un Bean, e che possono essere modificate durante tutto il ciclo di vita del componente. Di base, le proprietà sono attributi privati, ai quali si accede attraverso una coppia di metodi della forma:

```
public <PropertyType> get<PropertyName>()
public void set<PropertyName>(<PropertyType> property)
```

La convenzione di aggiungere il prefisso get e set ai metodi che forniscono l'accesso a una proprietà, permette ad esempio ai tool grafici di rilevare le proprietà Bean, determinarne le regole di accesso (Read Only o Read/Write), dedurne il tipo, visualizzare le proprietà su un apposito Property Sheet e individuare l'editor di proprietà più adatto al caso.

Se ad esempio un tool grafico scopre, grazie all'introspezione, la coppia di metodi

```
public Color getForegroundColor() { ... }
public void setForegroundColor(Color c) { ... }
```

da questi conclude che esiste una proprietà chiamata foregroundColor (notare la prima lettera minuscola), accessibile sia in lettura che in scrittura, di tipo Color. A questo punto, il tool può cercare un editor di proprietà per parametri di tipo Color, e mostrare la proprietà su un property sheet in modo che possa essere vista e manipolata dal programmatore.

## Proprietà indicizzate (Indexed Property)

Le proprietà indicizzate permettono di gestire collezioni di valori accessibili attraverso indice, in maniera simile a come si fa con un vettore. Lo schema di composizione dei metodi di accesso di una proprietà indicizzata è il seguente:

```
public <PropertyType>[] get<PropertyName>();
public void set<PropertyName>(<PropertyType>[] value);
```

per i metodi che permettono di manipolare l'intera collection, mentre per accedere ai singoli elementi, si deve predisporre una coppia di metodi del tipo:

```
public <PropertyType> get<PropertyName>(int index);
public void set<PropertyName>(int index, <PropertyType> value);
```

## Proprietà bound

Le proprietà semplici, così come sono state descritte nei paragrafi precedenti, seguono una convenzione radicata da tempo nella normale programmazione a oggetti. Le proprietà bound, al contrario, sono caratteristiche dell'universo dei componenti, dove si verifica molto spesso la necessità di collegare il valore delle proprietà di un componente a quelli di un'altro, in modo tale che si mantengano aggiornati. I metodi set delle proprietà bound, inviano una notifica a tutti gli ascoltatori registrati ogni qualvolta viene alterato il valore della proprietà. Il meccanismo di ascolto–notifica, simile a quello degli eventi Swing e AWT, segue il pattern Observer.

Le proprietà bound, a differenza degli eventi Swing, utilizzano un unico tipo di evento, ChangeEvent, cosa che semplifica il processo di implementazione. La classe PropertyChangeSupport, presente all'interno del package java.bean, fornisce i metodi che gestiscono la lista degli ascoltatori e quelli che producono l'invio degli eventi.

Un oggetto che voglia mettersi in ascolto di una proprietà, deve implementare l'interfaccia PropertyChangeListener e deve registrarsi presso la sorgente di eventi. L'oggetto PropertyChangeEvent incapsula le informazioni riguardo alla proprietà modificata, alla sorgente e al valore della proprietà.

**Figura 18.1 –** *Il meccanismo di notifica di eventi bound segue il pattern Observer*

## Come implementare il supporto alle proprietà bound

Per aggiungere a un Bean il supporto alle proprietà bound, bisogna anzitutto importare il package java.beans.*, in modo da garantire l'accesso alle classi PropertyChange-Support e PropertyChangeEvent. Quindi bisogna creare un oggetto PropertyChangeSupport, che ha il compito di mantenere la lista degli ascoltatori e di fornire i metodi che gestiscono l'invio degli eventi.

```
private PropertyChangeSupport changes = new PropertyChangeSupport(this);
```

A questo punto bisogna realizzare, nella propria classe, i metodi che permettono di gestire la lista degli ascoltatori. Tali metodi sono dei semplici metodi Wrapper che fanno riferimento a metodi con la stessa firma, presenti nel PropertyChangeSupport:

```
public void addPropertyChangeListener(PropertyChangeListener l) {
 changes.addPropertyChangeListener(l);
}
```

```
public void removePropertyChangeListener(PropertyChangeListener l) {
 changes.removePropertyChangeListener(l);
}
```

La presenza dei metodi addPropertyChangeListener() e removeProperty-ChangeListener() permette ai tool grafici di riconoscere un oggetto in grado di inviare proprietà bound e di mettere a disposizione un'opportuna voce nel menù di gestione degli eventi.

L'ultimo passaggio consiste nel modificare i metodi set relativi alle proprietà che si vuole rendere bound, per fare in modo che venga generato un PropertyChangeEvent ogni volta che la proprietà viene reimpostata

```
public void setColor(Color newColor) {
 Color oldColor = color;
 color = newColor;
 changes.firePropertyChange("color", oldColor , newColor);
}
```

Nel caso di proprietà read only, prive di metodo set, l'invio dell'evento dovrà avvenire all'interno del metodo che attua la modifica della proprietà. Un aspetto interessante del meccanismo di invio di PropertyChangeEvent, è che essi trasportano sia il nuovo valore che quello vecchio. Questa scelta dispensa chi implementa un ascoltatore dal compito di mantenere una copia del valore, qualora questo fosse necessario, dal momento che l'evento viene propagato *dopo* la modifica della relativa proprietà. Il metodo fireChangeEvent() della classe PropertyChangeListener fornisce il servizio di Event Dispatching:

```
firePropertyChange(String propertyName, Object oldValue, Object newValue)
```

In pratica esso impacchetta i parametri in un oggetto PropertyChangeEvent, e chiama il metodo propertyChange(PropertyChangeEvent p) su tutti gli ascoltatori registrati. I parametri vengono trattati come Object, e nel caso si debbano inviare proprietà espresse in termini di tipi primitivi, occorre incapsularle nell'opportuno Wrapper (Integer per valori int, Double per valori double e così via). Per facilitare questo compito, la classe propertyChangeSupport prevede delle varianti di firePropertyChange per valori int e boolean.

### Come implementare il supporto alle proprietà bound su sottoclassi di JComponent

La classe JComponent, superclasse di tutti i componenti Swing, dispone del supporto nativo alla gestione di proprietà bound. Di base essa fornisce i metodi addProperty-ChangeListener e removePropertyChangeListener, oltre a una collezione di metodi firePropertyChange adatta ad ogni tipo primitivo. In questo caso l'implementazione di una proprietà bound richiederà solo una modifica al metodo set preposto, similmente a come descritto nell'ultimo passaggio del precedente paragrafo, con la differenza che non è necessario ricorrere a un oggetto propertyChangeSupport per inviare la proprietà:

```
public void setColor(Color newColor) {
 Color oldColor = color;
 color = newColor;
 firePropertyChange("color", oldColor , newColor);
}
```

### Ascoltatori di proprietà

Se si desidera mettersi in ascolto di una proprietà, occorre definire un opportuno oggetto PropertyChangeListener e registrarlo presso il Bean. Un Property-ChangeListener deve definire il metodo propertyChange(PropertyChangeEvent e), che viene chiamato quando avviene la modifica di una proprietà bound.

Un PropertyChangeListener viene notificato quando avviene la modifica di *una qualunque* proprietà bound: per questa ragione esso deve, come prima cosa, verificare, che la proprietà appena modificata sia quella alla quale si è interessati. Una simile verifica richiede una chiamata al metodo getPropertyName di PropertyChangeEvent, che restituisce il nome della proprietà. Per convenzione, i nomi di proprietà vengono estratti dai nomi dichiarati nei metodi get e set, con la prima lettera minuscola. Il seguente frammento di codice presenta un tipico PropertyChange-Listener, che ascolta la proprietà foregroundColor:

```
public class Listener implements PropertyChangeListener() {
 public void propertyChange(PropertyChangeEvent e) {
   if(e.getPropertyName().equals("foregroundColor"))
     System.out.println(e.getNewValue());
 }
}
```

## Un esempio di Bean con proprietà bound

Un Java Bean rappresenta un mattone di un programma. Ogni componente è un'unità di utilizzo abbastanza grossa da incorporare una funzionalità evoluta, ma piccola rispetto ad un programma fatto e finito. Il concetto del riuso può essere presente a diversi livelli del progetto: il seguente Bean fornisce un esempio di elevata versatilità

Il Bean PhotoAlbum è un pannello grafico al cui interno vengono caricate delle immagini. Il metodo showNext() permette di passare da un'immagine all'altra, in modo ciclico. Il numero ed il tipo di immagini viene determinato al momento dell'avvio: durante la fase di costruzione viene letto il file comment.txt, presente nella directory images, che contiene una riga di commento per ogni immagine presente nella cartella. Le immagini devono essere nominate in modo progressivo (img0.jpg, img1.jpg, img2.jpg…) e devono essere presenti in numero uguale alle righe del file comment.txt. Questa scelta progettuale consente di introdurre il riuso a un livello abbastanza alto: qualunque utente, anche con scarse conoscenze del linguaggio, può personalizzare il componente, inserendo le sue foto preferite, senza la necessità di alterare il codice sorgente.

Il Bean PhotoAlbum ha tre proprietà:

- imageNumber, che restituisce il numero di immagini contenute nell'album. Essendo una quantità immutabile, tale proprietà è stata implementata come proprietà semplice.

- imageIndex: restituisce l'indice dell'immagine attualmente visualizzata. Al cambio di immagine viene inviato un PropertyChangeEvent.

- imageComment: restituisce una stringa di commento all'immagine. Anche in questo caso, al cambio di immagine viene generato un PropertyChangeEvent.

Il Bean viene definito come sottoclasse di JPanel: per questo motivo non vengono dichiarati i metodi addPropertyChangeListener e removeProperty-ChangeListener, già presenti nella superclasse. L'invio delle proprietà verrà messo in atto grazie al metodo firePropertyChange di JComponent.

```
package com.mokabyte.mokabook.javaBeans.photoAlbum;

import java.awt.*;
import java.beans.*;
import java.io.*;
import java.net.*;
import java.util.*;
import javax.swing.*;

public class PhotoAlbum extends JPanel {

 private Vector comments = new Vector();
 private int imageIndex;

 public PhotoAlbum() {
   super();
   setLayout(new BorderLayout());
   setupComments();
   imageIndex = 0;
   showNext();
 }
 private void setupComments() {
   try {
     URL indexUrl = getClass().getResource("images/" + "comments.txt");
     InputStream in = indexUrl.openStream();
     BufferedReader lineReader = new BufferedReader(new InputStreamReader(in));
     String line;
     while((line = lineReader.readLine())!=null)
       comments.add(line);
   }
   catch(Exception e) {
     e.printStackTrace();
```

```
    }
  }
  public int getImageNumber() {
    return comments.size();
  }
  public int getImageIndex() {
    return imageIndex;
  }
  public String getImageComment() {
    return (String)comments.elementAt(imageIndex);
  }
  public void showNext() {
    int oldImageIndex = imageIndex;
    imageIndex = ((imageIndex +1 ) % comments.size());
    String imageName = «img» + Integer.toString(imageIndex) + «.jpg»;
    showImage(getClass().getResource(„images/“ + imageName));
    String oldImageComment = (String)comments.elementAt(oldImageIndex);
    String currentImageComment = (String)comments.elementAt(imageIndex);
    firePropertyChange(“imageComment”, oldImageComment, currentImageComment);
    firePropertyChange(“imageIndex”, oldImageIndex, imageIndex);
  }
  private void showImage(URL imageUrl) {
    ImageIcon img = new ImageIcon(imageUrl);
    JLabel picture = new JLabel(img);
    JScrollPane pictureScrollPane = new JScrollPane(picture);
    removeAll();
    add(BorderLayout.CENTER,pictureScrollPane);
    validate();
  }
}
```

È possibile testare il Bean come fosse una normale classe Java, utilizzando queste semplici righe di codice:

```
package com.mokabyte.mokabook.javaBeans.photoAlbum;

import com.mokabyte.mokabook.javaBeans.photoAlbum.*;
import java.beans.*;
import javax.swing.*;

public class PhotoAlbumTest {
public static void main(String argv[]) {
  JFrame f = new JFrame(“Photo Album”);
  PhotoAlbum p = new PhotoAlbum();
  f.getContentPane().add(p);
  p.addPropertyChangeListener(new PropertyChangeListener() {
    public void propertyChange(PropertyChangeEvent e) {
```

```
            System.out.println(e.getPropertyName() + ": " + e.getNewValue());
        }
    });
  f.setSize(500,400);
  f.setVisible(true);

  while(true)
    for(int i=0;i<7;i++) {
          p.showNext();
          try {Thread.sleep(1000);}catch(Exception e) {}
        }
  }
  }
```

**Figura 18.2 –** *Un programma di prova per il Bean PhotoAlbum*

## Creazione di un file JAR

Prima di procedere alla consegna del Bean entro un file JAR, bisogna anzitutto compilare le classi PhotoAlbum.java e PhotoAlbumTest.java, che devono trovarsi nella cartella com\mokabyte\mokabook\javaBeans\

```
javac com\mokabyte\mokabook\javaBeans\photoAlbum\*.java
```

A questo punto bisogna creare, ricorrendo a un semplice editor di testo tipo Notepad, un file photoAlbumManifest.tmp con il seguente contenuto

```
Main-Class: com.mokabyte.mokabook.javaBeans.photoAlbum.PhotoAlbumTest
```

```
Name: com/mokabyte/mokabook/javaBeans/photoAlbum/PhotoAlbum.class
Java-Bean: True
```

Le prime due righe, opzionali, segnalano la presenza di una classe dotata di metodo main.

Le ultime due righe del file manifest specificano che la classe PhotoAlbum.class è un Java Bean. Se l'archivio contiene più di un Bean, è necessario elencarli tutti.

Per generare l'archivio photoAlbum.jar, bisogna digitare la riga di comando:

```
jar cfm photoAlbum.jar photoAlbumManifest.tmp
 com\mokabyte\mokabook\javaBeans\photoAlbum\*.class
 com\mokabyte\mokabook\javaBeans\photoAlbum\images\*.*
```

Il file così generato contiene tutte le classi e le immagini necessarie a dar vita al Bean PhotoAlbum. Tale file potrà essere utilizzato facilmente all'interno di tool grafici o di pagine web, racchiuso dentro una Applet.

Il file .jar potrà essere avviato digitando

```
java PhotoAlbum.jar
```

**Figura 18.3** – *Un file JAR opportunamente confezionato può essere aperto con un opportuno tool come Jar o WinZip*

Le istruzioni fornite sono valide per la piattaforma Windows. Su piattaforma Unix, le eventuali occorrenze del simbolo "\", che funge da path separator su piattaforme Windows, andranno sostituite col simbolo "/". Le convenzioni adottate all'interno del file manifest valgono invece su entrambe le piattaforme.

### Integrazione con altri Bean

Nonostante il Bean PhotoAlbum fornisca un servizio abbastanza evoluto, non è ancora classificabile come applicazione. Esso, opportunamente integrato con altri Beans, può comunque dar vita a numerosi programmi; di seguito, ecco qualche esempio: collegato a un CalendarBean, PhotoAlbum può dar vita a un simpatico calendario elettronico; collegando un bottone Bean al metodo showNext() è possibile creare un album interattivo, impacchettarlo su un'Applet e pubblicarlo su Internet; impacchettando il Bean PhotoAlbum con foto natalizie, e collegandolo con un Bean Carillon, si può ottenere un biglietto di auguri elettronico.

**Figura 18.4** – *Combinando, all'interno del Bean Box, il Bean PhotoAlbum con un pulsante Bean, si ottiene una piccola applicazione*

A questi esempi se ne possono facilmente aggiungere altri; altri ancora diventano possibili aggiungendo al Bean nuovi metodi, come previousImage() e setImageAt(int i); un compito ormai alla portata del lettore che fornisce un ottimo pretesto per esercitarsi.

## Eventi Bean

La notifica del cambiamento di valore delle proprietà bound è un meccanismo di comunicazione tra Beans. Se si vuole che un Bean sia in grado di propagare eventi di tipo più generico, o comunque eventi che non è comodo rappresentare come un cambiamento di stato, è possibile utilizzare un meccanismo di eventi generico, del tutto simile a quello pre-

sente nei componenti grafici Swing e AWT. I prossimi paragrafi servono a illustrare le tre fasi dell'implementazione: creazione dell'evento, definizione dell'ascoltatore e infine creazione della sorgente di eventi.

### Creazione di un evento

Per implementare un meccanismo di comunicazione basato su eventi, occorre anzitutto definire un'opportuna sottoclasse di EventObject, che racchiuda tutte le informazioni relative all'evento da propagare.

```
public class <EventType> extends EventObject {
 private <ParamType> param
 public <EventType>(Object source,<ParamType> param) {
   super(source);
   this.param = param;
 }
 public <ParamType> getParameter() {
   return param;
 }
}
```

La principale variazione sul tema si ha sul numero e sul tipo di parametri: tanto più complesso è l'evento da descrivere, maggiori saranno i parametri in gioco. L'unico parametro che è obbligatorio fornire è un reference all'oggetto che ha generato l'evento: tale reference, richiamabile con il metodo getSource() della classe EventObject, permetterà all'ascoltatore di interrogare la sorgente degli eventi qualora ce ne fosse bisogno.

### Destinatari di eventi

Il secondo passaggio è quello di definire l'interfaccia di programmazione degli ascoltatori di eventi. Tale interfaccia deve essere definita come sottoclasse di EventListener, per essere riconoscibile come ascoltatore dall'Introspector. Lo schema di sviluppo degli ascoltatori segue lo schema

```
import java.awt.event.*;

public Interface <EventListener> extends EventListener {
 public void <EventType>Performed(<EventType> e);
}
```

Le convenzioni di naming dei metodi dell'interfaccia non seguono uno schema standard: la convenzione descritta nell'esempio, <EventType>performed, può essere seguita o meno. L'importante è che il nome dei metodi dell'interfaccia Listener suggeriscano il tipo di azione sottostante, e che accettino come parametro un evento del tipo giusto.

## Sorgenti di eventi

Se si desidera aggiungere a un Bean la capacità di generare eventi, occorre implementare una coppia di metodi

```
add<EventListenerType>(<EventListenerType> l)
remove<EventListenerType>(<EventListenerType> l)
```

La gestione della lista degli ascoltatori e l'invio degli eventi segue una formula standard, descritta nelle righe seguenti:

```
private Vector listeners = new Vector();

public void add<EventListenerType>(<EventListenerType> l) {
 listeners.add(l);
}
public void remove<EventListenerType>(<EventListenerType> l) {
 listeners.remove(l);
}
protected void fire<Eventype>(<EvenType> e) {
 Enumeration listenersEnumeration = listeners.elements();
 while(listenersEnumeration.hasMoreElements()) {
 <EventListenerType> listener = (<EventListenerType>)listenersEnumeration.nextElement();
   listener.<EventType>Performed(e);
 }
}
```

## Sorgenti unicast

In alcuni casi occorre definire sorgenti di eventi capaci di servire un unico ascoltatore. Per implementare tali classi, che fungono da sorgenti unicast, si può seguire il seguente modello

```
private <EventListenerType> listener;

public void add<EventListenerType>(
<EventListenerType> l) throws TooManyListenersException {
 if(listener == null)
   listener = l;
 else
   throw new TooManyListenerException();
}
public void remove<EventListenerType>(<EventListenerType> l) {
 listener = null;
}
protected void fire<Eventype>(<EvenType> e) {
```

```
  if(listener! = null)
     listener.<EventType>Performed(e);
  }
```

## Ascoltatori di eventi: Event Adapter

Se si vuole che un evento generato da un Bean scateni un'azione su un altro Bean, è necessario creare un oggetto che realizzi un collegamento tra i due. Tale classe, detta Adapter, viene registrata come ascoltatore presso la sorgente dell'evento, e formula una chiamata al metodo destinazione ogni volta che riceve una notifica dal Bean sorgente.

Gli strumenti grafici tipo JBuilder generano questo tipo di classi in maniera automatica: tutto quello che l'utente deve fare è collegare, con pochi click di mouse, l'evento di un Bean sorgente a un metodo di un Bean target. Qui di seguito viene riportato il codice di un Adapter, generato automaticamente dal Bean Box, che collega la pressione di un pulsante al metodo `startJuggling(ActionEvent e)` del Bean `Juggler`.

```
// Automatically generated event hookup file.

public class ___Hookup_172935aa26 implements     java.awt.event.ActionListener, java.io.Serializable {

 public void setTarget(sunw.demo.juggler.Juggler t) {
    target = t;
 }

 public void actionPerformed(java.awt.event.ActionEvent arg0) {
    target.startJuggling(arg0);
 }

 private sunw.demo.juggler.Juggler target;
}
```

**Figura 18.5 –** *Un Adapter funge da ponte di collegamento tra gli eventi di un Bean e i metodi di un altro*

EventSource

+addEventListener()
+removeEventListener()
+fireEvent()

## Un esempio di Bean con eventi

Il prossimo esempio è un Bean Timer, che ha il compito di generare battiti di orologio a intervalli regolari. Questo componente è un tipico esempio di Bean non grafico.

La prima classe che si definisce è quella che implementa il tipo di evento

```
package com.mokabyte.mokabook.javaBeans.timer;

import com.mokabyte.mokabook.javaBeans.*;
import java.io.*;
import java.util.*;

public class TimerEvent extends EventObject implements Serializable {

 public TimerEvent(Object source) {
   super(source);
 }
}
```

Come si può vedere, l'implementazione di un nuovo tipo di evento è questione di poche righe di codice. L'unico particolare degno di nota è che il costruttore del nuovo tipo di evento deve invocare il costruttore della superclasse, passando un reference alla sorgente dell'evento.

L'interfaccia che rappresenta l'ascoltatore deve estendere l'interfaccia EventListener; a parte questo, al suo interno si può definire un numero arbitrario di metodi, la cui unica costante è quella di avere come parametro un reference all'evento da propagare.

```
package com.mokabyte.mokabook.javaBeans.timer;

import com.mokabyte.mokabook.javaBeans.timer.*;

public interface TimerListener extends java.util.EventListener {
 public void clockTicked(TimerEvent e);
}
```

Per finire, ecco il Bean vero e proprio. Come si può notare, esso implementa l'interfaccia Serializable che rende possibile la serializzazione.

```
package com.mokabyte.mokabook.javaBeans.timer;

import com.mokabyte.mokabook.javaBeans.*;
import java.io.*;
import java.util.*;

public class TimerBean implements Serializable {
```

```
private int time = 1000;
private transient TimerThread timerThread;
private Vector timerListeners = new Vector();

public void addTimerListener(TimerListener t) {
 timerListeners.add(t);
}
public void removeTimerListener(TimerListener t) {
 timerListeners.remove(t);
}
protected void fireTimerEvent(TimerEvent e) {
 Enumeration listeners = timerListeners.elements();
 while(listeners.hasMoreElements())
    ((TimerListener)listeners.nextElement()).clockTicked(e);
}
public synchronized void setMillis(int millis) {
 time = millis;
}
public synchronized int getMillis() {
 return time;
}
public synchronized void startTimer() {
 if(timerThread!=null)
    forceTick();
 timerThread = new TimerThread();
 timerThread.start();
}
public synchronized void stopTimer() {
 if(timerThread == null)
    return;

 timerThread.killTimer();
 timerThread = null;
}
public synchronized void forceTick() {
 if(timerThread!=null) {
    stopTimer();
    startTimer();
 }
 else
    fireTimerEvent(new TimerEvent(this));
}

 class TimerThread extends Thread {
    private boolean running = true;

    public synchronized void killTimer() {
```

```
            running = false;
        }
        private synchronized boolean isRunning() {
            return running;
        }
        public void run() {
            while(true)
                try {
                    if(isRunning()) {
                        fireTimerEvent(new TimerEvent(TimerBean.this));
                        Thread.sleep(getMillis());
                    }
                    else
                        break;
            catch(InterruptedException e) {}
            }
        }
    }
}
```

I primi tre metodi servono a gestire la lista degli ascoltatori. Il terzo e il quarto gestiscono la proprietà millis, ossia il tempo, in millisecondi, tra un tick e l'altro. I due metodi successivi, startTimer, stopTimer, servono ad avviare e fermare il timer, mentre forceTick lancia un tick e riavvia il timer, se questo è attivo. Il timer vero e proprio viene implementato grazie a una classe interna TimerThread, sottoclasse di Thread. Si noti il metodo killTimer, che permette di terminare in modo pulito la vita del thread: questa soluzione è da preferire al metodo stop (deprecato a partire dal JDK 1.1), che in certi casi può provocare la terminazione del thread in uno stato inconsistente.

Per compilare le classi del Bean, bisogna usare la seguente riga di comando

```
javac com\mokabyte\mokabook\javaBeans\timer\*.java
```

Per impacchettare il Bean in un file .jar, è necessario per prima cosa creare con un editor di testo il file timerManifest.tmp, con le seguenti righe

```
Name: com/mokabyte/mokabook/javaBeans/timer/TimerBean.class
Java-Bean: True
```

Per creare l'archivio si deve quindi digitare il seguente comando

```
jar cfm timer.jar timerManifest.tmp com\mokabyte\mokabook\javaBeans\timer\*.class
```

Per testare la classe TimerBean, si può usare il seguente programma, che crea un oggetto TimerBean e registra un TimerListener il quale stampa a video una scritta ad ogni tick del timer.

```
package com.mokabyte.mokabook.javaBeans.timer;

import com.mokabyte.mokabook.javaBeans.timer.*;

public class TimerTest {
 public static void main(String argv[]) {


TimerBean t = new TimerBean();
   t.addTimerListener(new TimerListener() {
      public void clockTicked(TimerEvent e) {
         System.out.println("Tick");
      }
   });
   t.startTimer();
 }
}
```

# Introspezione: l'interfaccia BeanInfo

Le convenzioni di naming descritte nei paragrafi precedenti permettono ai tool grafici abilitati ai Beans di scoprire i servizi di un componente grazie alla reflection. Questo processo automatico è certamente comodo, ma ha il difetto di non offrire nessun tipo di controllo sul numero e sul tipo di servizi da mostrare. In alcune occasioni può essere necessario mascherare un certo numero di servizi, specie quelli ereditati da una superclasse.

I Beans creati a partire dalla classe `JComponent`, ad esempio, ereditano automaticamente più di dieci attributi (dimensioni, colore, allineamento…) e ben dodici tipi diversi di evento (`ComponentEvent`, `MouseEvent`, `HierarchyEvent`…). Un simile eccesso provoca di solito disorientamento nell'utente; in questi casi è preferibile fornire un elenco esplicito dei servizi da associare al nostro Bean, in modo da "ripulire" gli eccessi.

Per raggiungere questo obiettivo, bisogna associare al Bean una classe di supporto, che implementi l'interfaccia `BeanInfo`. Una classe `BeanInfo` permette di fare un certo numero di cose: esporre solamente i servizi che si desidera rendere visibili, aggirare le convenzioni di naming imposte dalle specifiche Java Beans, associare al Bean un'icona e attribuire ai servizi nomi più descrittivi di quelli rilevabili con il processo di analisi delle firme dei metodi.

## Creare una classe BeanInfo

Per creare una classe `BeanInfo` bisogna anzitutto definire una classe con lo stesso nome del Bean, a cui si deve aggiungere il suffisso `BeanInfo`. Per semplificare il lavoro si può estendere `SimpleBeanInfo`, una classe che fornisce un'implementazione nulla di tutti i metodi dell'interfaccia. In questo modo ci si limiterà a sovrascrivere solamente i metodi che interessano, lasciando tutti gli altri con l'impostazione di default.

Per ridefinire il numero ed il tipo dei servizi Bean, occorre agire in modo appropriato a restituire le proprietà, i metodi o gli eventi che si desidera esporre. Opzionalmente, si può associare

un'icona al Bean, definendo il metodo public java.awt.Image getIcon(int iconKind). Per finire, si può specificare la classe del Bean e il suo Customizer, qualora ne esista uno, con il metodo public BeanDescriptor getBeanDescriptor().

La classe BeanInfo così prodotta deve essere messa nello stesso package che contiene il Bean. In assenza di una classe BeanInfo, i servizi di un Bean vengono trovati con la reflection.

### Feature Descriptors

Una classe di tipo BeanInfo restituisce, tramite i seguenti metodi, vettori di *descriptors* che contengono informazioni relative ad ogni proprietà, metodo o evento che il progettista di un Bean desidera esporre.

```
PropertyDescriptor[] getPropertyDescriptors();
MethodDescriptor[] getMethodDescriptors();
EventSetDescriptor[] getEventSetDescriptors();
```

Ogni Descriptor fornisce una precisa rappresentazione di una classe di servizi Bean. Il package java.bean implementa le seguenti classi:

- FeatureDescriptor: è la classe base per tutte le altre classi Descriptor, e definisce gli aspetti comuni a tutta la famiglia.
- BeanDescriptor: descrive il tipo e il nome della classe Bean associati, oltre a fornire il Customizer, se ne esiste uno.
- PropertyDescriptor: descrive le proprietà del Bean.
- IndexedPropertyDescriptor: è una sottoclasse di PropertyDescriptor, e descrive le proprietà indicizzate.
- EventSetDescriptor: descrive gli eventi che il Bean è in grado di inviare.
- MethodDescriptor: descrive i metodi del Bean.
- ParameterDescriptor: descrive i parametri dei metodi.

## Esempio

In questo esempio si analizzerà un BeanInfo per il Bean PhotoAlbum, che permette di nascondere una grossa quantità di servizi Bean che per default vengono ereditati dalla superclasse JPanel.

```
package com.mokabyte.mokabook.javaBeans.photoAlbum;

import java.beans.*;
```

```
import com.mokabyte.mokabook.javaBeans.photoAlbum.*;

public class PhotoAlbumBeanInfo extends SimpleBeanInfo {

private static final Class beanClass = PhotoAlbum.class;

 public PropertyDescriptor[] getPropertyDescriptors() {
   try {
     PropertyDescriptor imageNumber
     = new PropertyDescriptor("imageNumber", beanClass, "getImageNumber", null);
     PropertyDescriptor imageIndex = new PropertyDescriptor("imageIndex", beanClass, "getImageIndex", null);
     PropertyDescriptor imageComment
     = new PropertyDescriptor("imageComment", beanClass, "getImageComment", null);

     imageIndex.setBound(true);
     imageComment.setBound(true);

     PropertyDescriptor properties[]
     = {imageNumber, imageIndex, imageComment};
     return properties;
   } catch (IntrospectionException e) {
      throw new Error(e.toString());
   }
 }
 public EventSetDescriptor[] getEventSetDescriptors() {
   try {
     EventSetDescriptor changed
     = new EventSetDescriptor(beanClass, "propertyChange", PropertyChangeListener.class, "propertyChange");
     changed.setDisplayName("Property Change");
     EventSetDescriptor events[] = {changed};
     return events;
   } catch (IntrospectionException e) {
     throw new Error(e.toString());
   }
 }
 public MethodDescriptor[] getMethodDescriptors() {
   try {
     MethodDescriptor showNext
     = new MethodDescriptor(beanClass.getMethod("showNext", null));

       MethodDescriptor methods[] = {showNext};
       return methods;
   } catch (Exception e) {
        throw new Error(e.toString());
   }
 }
 public java.awt.Image getIcon(int iconKind){
```

```
        if(iconKind == SimpleBeanInfo.ICON_COLOR_16x16)
          return loadImage("photoAlbumIcon16.gif");
        else
          return loadImage("photoAlbumIcon32.gif");
    }
  }
```

La classe viene definita come sottoclasse di SimpleBeanInfo, in modo da rendere il processo di sviluppo più rapido.

Il primo metodo, getPropertyDescriptors, restituisce un array con un tre PropertyDescriptor, uno per ciascuna delle proprietà che si vogliono rendere visibili. Il costruttore di PropertyDescriptor richiede quattro argomenti: il nome della proprietà, la classe del Bean, il nome del metodo getter e quello del metodo setter: quest'ultimo è posto a null, a significare che le proprietà sono di tipo Read Only. Si noti, in questo metodo e nei successivi, che la creazione dei Descriptors deve essere definita all'interno di un blocco try–catch, dal momento che può generare IntrospectionException.

Il secondo metodo, getEventSetDescriptors(), restituisce un vettore con un unico EventSetDescriptor. Quest'ultimo viene inizializzato con quattro parametri: la classe del Bean, il nome della proprietà, la classe dell'ascoltatore e la firma del metodo che riceve l'evento. Si noti la chiamata al metodo setDisplayName(), che permette di impostare un nome più leggibile di quello che viene normalmente ottenuto dalle firme dei metodi.

Il terzo metodo, getMethodDescriptors, restituisce un vettore contenente un unico MethodDescriptor, che descrive il metodo showNext(). Il costruttore di MethodDescriptor richiede come unico parametro un oggetto di classe Method, che in questo esempio viene richiesto alla classe PhotoAlbum ricorrendo alla reflection.

Infine il metodo getIcon() restituisce un'icona, che normalmente viene associata al Bean all'interno di strumenti visuali.

Per impacchettare il Bean PhotoAlbum con le icone e il BeanInfo, si può seguire la procedura già descritta, modificando la riga di comando dell'utility jar in modo da includere le icone nell'archivio.

```
jar cfm photoAlbum.jar photoAlbumManifest.tmp com\mokabyte\mokabook\
javaBeans\photoAlbum\*.class com\mokabyte\mokabook\javaBeans\
photoAlbum\*.gif com\mokabyte\mokabook\javaBeans\photoAlbum\images\*.*
```

## Personalizzazione dei Bean

L'aspetto e il comportamento di un Bean possono essere personalizzati in fase di composizione all'interno di un tool grafico abilitato ai Beans. Esistono due strumenti per personalizzare un Bean: gli Editor di proprietà e i Customizer. Gli Editor di proprietà sono componenti grafici specializzati nell'editing di un particolare *tipo* di proprietà: interi, stringhe, files… Ogni Editor di proprietà viene associato a un particolare tipo Java, e il tool grafico compone automaticamente un Property Sheet analizzando le proprietà di un Bean, e ricorrendo agli Editor più adatti alla circostanza. In fig. 18.6 si può vedere un esempio di Property Sheet, realizzato dal Bean Box: ogni riga presenta, accanto al nome della proprietà, il relativo Editor.

**Figura 18.6 –** *Un Property Sheet generato in modo automatico a partire dalle proprietà di un pulsante Bean*

**Figura 18.7 –** *Il Property Sheet relativo a un pulsante Bean. Si noti il pannello ausiliario FontEditor*

Un Customizer, d'altra parte, è un pannello di controllo specializzato per un particolare Bean: in questo caso è il programmatore a decidere cosa mostrare nel pannello e in quale maniera. Per questa ragione un Customizer viene associato, grazie al BeanInfo, a un particolare Bean e non può, in linea di massima, essere usato su Bean differenti.

## Come creare un Editor di proprietà

Un Editor di proprietà deve implementare l'interfaccia PropertyEditor, o in alternativa, estendere la classe PropertyEditorSupport che fornisce un'implementazione standard dei metodi dell'interfaccia. L'interfaccia PropertyEditor dispone di metodi che permettono di specificare come una proprietà debba essere rappresentata in un property sheet. Alcuni Editor consistono in uno strumento direttamente editabile, altri presentano uno strumento a scelta multipla, come un ComboBox; altri ancora, per permettere la modifica, aprono un pannello separato, come nella proprietà font dell'esempio, che viene modificata grazie al pannello ausiliario FontEditor.

Per fornire il supporto a queste modalità di editing, bisogna implementare alcuni metodi di PropertyEditor, in modo che ritornino valori non nulli.

I valori numerici o String possono implementare il metodo setAsText(String s), che estrae il valore della proprietà dalla stringa che costituisce il parametro. Questo sistema permette di inserire una proprietà con un normale campo di testo.

Gli Editor standard per le proprietà Color e Font usano un pannello separato, e ricorrono al Property Sheet solamente per mostrare l'impostazione corrente. Facendo click sul valore, viene aperto l'Editor vero e proprio. Per mostrare il valore corrente della proprietà, è necessario sovrascrivere il metodo isPaintable() in modo che restituisca true, e sovrascrivere paintValue in modo che dipinga la proprietà attuale in un rettangolo all'interno del Property Sheet.

Per supportare l'Editor di Proprietà personalizzato occorre sovrascrivere altri due metodi della classe PropertyEditorSupport: supportsCustomEditor, che in questo caso deve restituire true, e getCustomEditor, in modo che restituisca un'istanza dell'Editor.

## Registrare gli Editor

I Property Editor vengono associati alle proprietà attraverso un'associazione esplicita, all'interno del metodo getPropertyDescriptors() del BeanInfo, con una chiamata al metodo setPropertyEditorClass(Class propertyEditorClass) del PropertyDescriptor corrispondente, come avviene nel Bean Molecule

```
PropertyDescriptor pd = new PropertyDescriptor("moleculeName", Molecule.class);
pd.setPropertyEditorClass(MoleculeNameEditor.class);
```

**Figura 18.8 –** *Il Bean Molecule associa alla proprietà moleculeName di un Editor di proprietà personalizzato*

In alternativa si può registrare l'Editor con il seguente metodo statico

```
PropertyEditorManager.registerEditor(Class targetType, Class editorType)
```

che richiede come parametri la classe che specifica il tipo e quella che specifica l'Editor.

## Customizers

Con un Bean Customizer è possibile fornire un controllo completo sul modo in cui configurare ed editare un Bean. Un Customizer è in pratica una piccola applicazione specializzata nell'editing di un particolare Bean, ogni volta che la configurazione di un Bean richiede modalità troppo sofisticate per il normale processo di creazione automatica del Property Sheet.

Le uniche regole a cui ci si deve attenere per realizzare un Customizer sono:

- deve estendere la classe `Component`, o una delle sue sottoclassi;
- deve implementare l'interfaccia `java.bean.Customizer`;
- deve implementare un costruttore privo di parametri.

Per associare il Customizer al proprio Bean, bisogna sovrascrivere il metodo `getBeanDescriptor` nella classe `BeanInfo`, in modo che restituisca un opportuno `BeanDescriptor`, il quale a sua volta dovrà restituire la classe del Customizer alla chiamata del metodo `getCustomizerClass`.

## Serializzazione

Per rendere serializzabile una classe Bean è di norma sufficiente implementare l'interfaccia `Serializable`, sfruttando così l'Object Serialization di Java. L'interfaccia `Serializable` non contiene metodi: essa viene usata dal compilatore per marcare le classi che possono essere serializzate. Esistono solo poche regole per implementare classi `Serializable`: anzitutto è necessario dichiarare un costruttore privo di argomenti, che verrà chiamato quando l'oggetto verrà ricostruito a partire da un file `.ser`; in secondo luogo una classe serializzabile deve definire al suo interno solamente attributi serializzabili.

Se si desidera fare in modo che un particolare attributo non venga salvato al momento della serializzazione, si può ricorrere al modificatore `transient`. La serializzazione standard, inoltre, non salva lo stato delle variabili `static`.

Per tutti i casi in cui la serializzazione standard non risultasse applicabile, occorre procedere all'implementazione dell'interfaccia `Externalizable`, fornendo, attraverso i metodi `readExternal(ObjectInput in)` e `writeExternal(ObjectOutput out)`, delle istruzioni esplicite su come salvare lo stato di un oggetto su uno stream e come ripristinarlo in un secondo tempo.

# Installazione dell'SDK

Giovanni Puliti

## Scelta del giusto SDK

Per poter lavorare con applicazioni Java o crearne di nuove, il programmatore deve poter disporre di un ambiente di sviluppo e di esecuzione compatibile con lo standard 100% Pure Java. Sun da sempre rilascia un kit di sviluppo che contiene tutti gli strumenti necessari per la compilazione ed esecuzione di applicazioni Java. Tale kit è comunemente noto come Java Development Kit (JDK): nel corso del tempo sono state rilasciate le versioni 1.0, 1.1, 1.2, 1.3 e 1.4. Attualmente l'ultima versione rilasciata è la 1.4, mentre si annuncia un prossimo JDK 1.5. Il JDK comprende una Java Virtual Machine (JVM), invocabile con il comando `java`, un compilatore (comando `javac`), un debugger (`jdbg`), un interprete per le applet (`appletviewer`) e altro ancora.

A partire dalla versione 1.2, Sun ha introdotto una nomenclatura differente per le varie versioni del kit di sviluppo. In quel momento nasceva infatti Java 2, a indicare la raggiunta maturità del linguaggio e della piattaforma. Pur mantenendo la completa compatibilità con il passato, Java 2 ha introdotto importanti miglioramenti, quali una maggiore stabilità e sicurezza, migliori performance e l'ottimizzazione dell'uso della memoria.

Con Java 2 nasce il concetto di SDK: non più un Java Development Kit ma un Software Development Kit. Il linguaggio Java può essere finalmente considerato un potente strumento general purpose.

La notazione di JDK non è stata eliminata: il JDK è formalmente una release dell'SDK Java 2.

Con Java 2, per organizzare e raccogliere al meglio le diverse tecnologie che costituiscono ormai la piattaforma, Sun ha suddiviso l'SDK in tre grandi categorie:

- Java 2 Standard Edition (J2SE): questa versione contiene la JVM standard più tutte le librerie necessarie per lo sviluppo della maggior parte delle applicazioni Java.

- Java 2 Enterprise Edition (J2EE): contiene in genere le API enterprise come EJB, JDBC 2.0, Servlet ecc. La JVM normalmente è la stessa, quindi lavorare direttamente con l'SDK in boundle spesso non è molto utile: è molto meglio partire dalla versione J2SE e aggiungere l'ultima versione delle API EE, a seconda delle proprie esigenze.

- Java 2 Micro Edition (J2ME): Java è nato come linguaggio portabile in grado di essere eseguito con ogni tipo di dispositivo. La J2ME include una JVM e un set di API e librerie appositamente limitate, per poter essere eseguite su piccoli dispositivi embedded, telefoni cellulari ed altro ancora. Questa configurazione deve essere scelta solo se si vogliono scrivere applicazioni per questo genere di dispositivi.

La procedura di installazione è in genere molto semplice, anche se sono necessarie alcuni piccoli accorgimenti per permettere un corretto funzionamento della JVM e dei programmi Java. Si limiterà l'attenzione alla distribuzione J2SE. Per chi fosse interessato a scrivere programmi J2EE non vi sono particolari operazioni aggiuntive da svolgere. Per la J2ME, invece, il processo è del tutto analogo, anche se si deve seguire un procedimento particolare a seconda del dispositivo scelto e della versione utilizzata.

I file per l'installazione possono essere trovati direttamente sul sito di Sun, come indicato in [SDK]. Altri produttori rilasciano JVM per piattaforme particolari (molto note e apprezzate sono quelle di IBM). Per la scelta di JVM diverse da quelle prodotte da Sun si possono seguire le indicazioni della casa produttrice o del particolare sistema operativo utilizzato.

Chi non fosse interessato a sviluppare applicazioni Java, ma solo a eseguire applicazioni già finite, potrà scaricare al posto dell'SDK il Java Runtime Environment (JRE) , che in genere segue le stesse edizioni e release dell'SDK. Non sempre il JRE è sufficiente: per esempio, se si volessero eseguire applicazioni JSP già pronte, potrebbe essere necessario mettere ugualmente a disposizione di Tomcat (o di un altro servlet-JSP engine) un compilatore Java, indispensabile per la compilazione delle pagine JSP.

# Installazione su Windows

In ambiente Windows, in genere, il file di installazione è un eseguibile autoinstallante, che guida l'utente nelle varie fasi della procedura.

Non ci sono particolari aspetti da tenere in considerazione, a parte la directory di installazione e le variabili d'ambiente da configurare. Per la prima questione, a volte l'installazione nella directory "program files" può causare problemi di esecuzione ad alcuni applicativi Java che utilizzano la JVM di sistema (per esempio Tomcat o JBoss). Per questo, si consiglia di installare in una directory con un nome unico e senza spazi o altri caratteri speciali ("c:\programs", "c:\programmi" o semplicemente "c:\java").

Per poter funzionare, una qualsiasi applicazione Java deve sapere dove è installato il JDK, e quindi conoscere il path dell'eseguibile java (l'interprete), di javac (il compilatore usato da Tomcat per compilare le pagine JSP) e di altri programmi inclusi nel JDK.

Inoltre, un programma Java deve anche poter ricavare la variabile d'ambiente CLASSPATH, all'interno della quale dovranno essere inseriti i riferimenti ai vari package utilizzati (directory scompattate, file .jar o .zip). Al momento dell'installazione, il classpath viene automaticamente impostato in modo da contenere le librerie di base del Java SDK (in genere, nella sottodirectory jre/lib, o semplicemente lib).

A partire dal JDK 1.1 è invalsa l'abitudine di utilizzare la variabile JAVA_HOME, che deve puntare alla directory di installazione del JDK. Di conseguenza, il path di sistema dovrà essere impostato in modo che punti alla directory %JAVA_HOME%\bin.

Di norma, queste impostazioni sono effettuate in modo automatico dal programma di installazione, ma possono essere facilmente modificate o impostate ex-novo tramite il pannello di controllo di Windows.

Per esempio, aprendo la finestra per l'impostazione delle variabili d'ambiente dal pannello di controllo...

**Figura A.1 –** *In Windows il pannello di controllo permette di impostare le diverse variabili d'ambiente.*

...si può procedere a inserire sia la variabile JAVA_HOME (in questo caso c:\programs\jdk1.4) sia la directory con gli eseguibili nel path (%JAVA_HOME%\bin).

**Figura A.2 –** *Come impostare la variabile JAVA_HOME in modo che punti alla directory di installazione del JDK.*

**Figura A.3 –** *Come impostare il path in modo che includa la dir* JAVA_HOME\bin.

Se tutto è stato fatto come si deve, aprendo una console DOS si può verificare la correttezza delle impostazioni inserite.

Per esempio, per conoscere il contenuto della variabile JAVA_HOME si potrà scrivere:

```
C:\>echo %JAVA_HOME%
C:\programs\jdk1.4
```

Il comando path, invece, mostrerà Tra le altre cose:

```
C:\> path
PATH=…….C:\programs\jdk1.4\bin
```

A questo punto, si può provare a eseguire la JVM con il comando:

```
C:\>java -version
```

L'opzione –version permette di conoscere la versione della JVM installata. In questo caso, il comando restituisce il seguente output:

```
java version "1.4.1"
Java(TM) 2 Runtime Environment, Standard Edition (build 1.4.1-b21)
Java HotSpot(TM) Client VM (build 1.4.1-b21, mixed mode)
```

se si usa un sistema basato su Windows 95-98 o ME, l'impostazione delle variabili d'ambiente può essere fatta tramite il file autoexec.bat. In questo caso, con un'istruzione del tipo:

set JAVA_HOME="…."

Si potrà definire la variabile in modo che punti alla directory indicata. Si tenga presente che tali sistemi operativi offrono un supporto ridotto per lo sviluppo e l'esecuzione di applicazioni che fanno uso dei protocolli di rete (socket TCP, database ecc...) o server side (servlet, web application, EJB, per esempio). Si consiglia pertanto di utilizzare le versioni più evolute (NT, 2000, XP), che supportano in modo più corretto e completo lo strato di network TCP/IP e offrono maggiori servizi.

# Installazione su Linux

Per l'installazione su Linux, la procedura è molto simile a quella per Windows: si deve scaricare un file di installazione e installarlo. Per quest'ultimo aspetto si può utilizzare un rpm autoinstallante o un file eseguibile con estensione .bin.

Per esempio, se j2sdk-1.4.2-nb-3.5-bin-linux.bin è il file installante scaricato dal sito Sun, per prima cosa lo si renda eseguibile con il comando:

```
chmod o+x j2sdk-1.4.2-nb-3.5-bin-linux-i586.bin
```

quindi lo si mandi in esecuzione tramite:

```
./j2sdk-1.4.2-nb-3.5-bin-linux-i586.bin
```

per eseguire l'installazione. Di norma, questo porterà all'installazione dell'SDK in una directory il cui nome segue lo schema usr/java/jdk-<version-number>.

Questo significa che dovranno essere modificate di conseguenza le variabili JAVA_HOME e PATH, intervenendo sui file di profilo .bashrc o .bash_properties (a seconda del tipo di shell usata) dell'utente che dovrà usare Java:

```
JAVA_HOME=/usr/java/jdk1.4.1/
export JAVA_HOME
PATH=$JAVA_HOME/bin:$PATH
export PATH
```

Nel caso in cui un'applicazione debba far uso di altri package oltre a quelli di base del Java SDK, come un parser XML Xerces (contenuto in genere in un file xerces.jar), il package Java Mail, la Servlet API o altro ancora, si dovrà aggiungere manualmente al classpath il contenuto di tali librerie. Questo può essere fatto in due modi.

Il primo sistema consiste nell'aggiungere tali librerie al classpath di sistema, tramite il pannello di controllo di Windows o mediante l'impostazione ed esportazione di una variabile globale su Linux. In questo caso si potrà essere sicuri che tutte le applicazioni che dovranno utilizzare un parser Xerces o JavaMail potranno funzionare correttamente senza ulteriori impostazioni.

Attualmente, però, lo scenario Java è molto complesso, quindi un'impostazione globale difficilmente si adatta a tutte le applicazioni: in un caso potrebbe essere necessaria la versione 1.0 di Xerces, mentre un'altra applicazione potrebbe funzionare solo con la 1.2. Per questo motivo, in genere, si preferisce impostare un classpath personalizzato per ogni applicazione, passando tale configurazione alla JVM con il flag –classpath o –cp. Per esempio, in Windows si potrebbe scrivere:

```
set MY_CP=c:\programs\mokabyte\mypackages.jar
java –cp %MY_CP% com.mokabyte.mokacode.TestClasspathApp
```

Dove TestClasspathApp potrebbe essere un'applicazione che abbia bisogno di una serie di classi e interfacce contenute in mypackages.jar.

In questo modo si potranno costruire tutti i classpath personalizzati, concatenando file e directory di vario tipo.

In ambito J2EE le cose si complicano: entrano infatti in gioco il tipo di applicazione e le regole di caricamento del classloader utilizzato. Per questi aspetti, che comunque riguardano il programmatore esperto, si rimanda alla documentazione del prodotto utilizzato, e si consiglia l'adeguamento alle varie convenzioni imposte dalla specifica Java.

# Bibliografia

[SUN] – Sito web ufficiale di Sun dedicato a Java: http://java.sun.com

[SDK] – Sito web di Sun per il download dell'SDK nelle versioni per le varie piattaforme: http://java.sun.com/downloads

[JIBM] – Sito web IBM dedicato a Java: http://www.ibm.com/java

[xIBM] – "IBM Developer Kit for Linux: Overview": http://www-106.ibm.com/developerworks/java/jdk/linux140/?dwzone=java

# Ginipad, un ambiente di sviluppo per principianti

ANDREA GINI

Ginipad è un ambiente di sviluppo per Java realizzato da Andrea Gini, uno degli autori di questo manuale. Ginipad è stato pensato come strumento per principianti, che non hanno tempo o voglia di barcamenarsi tra editor testuali e tool a riga di comando. La sua interfaccia grafica semplice ed essenziale ne ha decretato il successo anche presso utenti più esperti, che spesso necessitano di uno strumento rapido e leggero da alternare agli ambienti di sviluppo più complessi.

Ginipad è stato progettato per offrire il massimo grado di funzionalità nel modo più semplice e intuitivo possibile. Bastano cinque minuti per prendere confidenza con l'ambiente e le sue funzioni. Questa appendice fornisce una tabella riassuntiva dei principali comandi e una guida all'installazione.

Si consiglia di visitare la home page del progetto per trovare tutte le informazioni necessarie:

http://www.mokabyte.it/ginipad

Il tempo di apprendimento può essere ridotto ad appena cinque minuti grazie a uno slideshow in PowerPoint, disponibile all'indirizzo:

http://www.mokabyte.it/ginipad/download/GinipadVisualTutorial.ppt

# Caratteristiche principali

**Figura B.1 –** *Caratteristiche principali di Ginipad.*

1. Pochi pulsanti facili da identificare.
2. Possibilità di lavorare su più di un documento.
3. Editor con Syntax Highlight.
4. Indice navigabile di metodi, campi e classi interne.
5. Autocompletamento delle dichiarazioni.
6. Hyperlink verso gli errori.

# Tabella riassuntiva dei comandi

| **File** | | | |
|---|---|---|---|
| | New | (Ctrl-N) | Crea un nuovo sorgente Java. |
| | Open | (Ctrl-O) | Carica un sorgente da disco. |
| | Save As | (Ctrl-S) | Salva il documento corrente. |
| | Close | (Ctrl-W) | Chiude il documento corrente. |
| | Open All | | Apre in una volta sola gli ultimi otto sorgenti. |
| | Exit | | Chiude il programma. |
| **Edit** | | | |
| | Cut | (Ctrl-X) | Taglia il testo selezionato. |
| | Copy | (Ctrl-C) | Copia il testo selezionato. |
| | Paste | (Ctrl-V) | Incolla il testo contenuto nella clipboard. |
| | Select All | (Ctrl-A) | Seleziona tutto il contenuto dell'editor. |
| | Undo | (Ctrl-Z) | Annulla l'ultima modifica. |
| | Redo | (Ctrl-Y) | Ripristina l'ultima modifica. |
| | Find | (Ctrl-F) | Apre la finestra di dialogo Find. |
| | Replace | (Ctrl-R) | Apre la finestra di dialogo Replace. |
| **Tools** | | | |
| | Compile | (Ctrl-Shift-C) | Compila il documento corrente. |
| | Run | (Ctrl-Shift-R) | Esegue il documento corrente. |
| | Stop | | Interrompe l'esecuzione del processo corrente. |
| | Format source code | (Ctrl-Shift-F) | Esegue una formattazione del codice. |
| **Console** | | | |
| | Hide Tree | | Nasconde il componente ad albero. |
| | Show Tree | | Mostra il componente ad albero. |
| | Hide Console | | Nasconde la console. |
| | Show Console | | Mostra la console. |
| | Show Panels | | Mostra tutti i pannelli. |

| | Full Screen | | Espande l'editor a pieno schermo. |
|---|---|---|---|
| | Clear Console | | Ripulisce la console. |
| **Dialog** | | | |
| | Preferences | | Apre la finestra delle preferenze. |
| | Help | | Apre la finestra di Help. |
| | About | | Apre la finestra di About. |

# Installazione

1. Caricare e Installare il JDK 1.4.
2. Lanciare il file di setup di Ginipad.
3. Al secondo passaggio della fase di installazione verrà richiesto di scegliere la Virtual Machine. Si consiglia di scegliere quella presente nella cartella \bin del JDK, come si vede in Fig B.2.

**Figura B.2 –** *Scelta della Virtual Machine.*

4. Al termine dell'installazione si può avviare il programma. Al primo avvio, Ginipad effettua una ricerca per localizzare la posizione del JDK. Tale processo è automatico e trasparente all'utente.

## Cosa fare se Ginipad non trova il JDK

Ginipad è in grado di identificare da solo la posizione del JDK su disco, durante la fase di installazione. Tuttavia, se tale posizione dovesse cambiare, per esempio in seguito a un aggiornamento del JDK, all'avvio successivo verrà richiesto di indicare la nuova posizione dell'ambiente di sviluppo.

**Figura B.3 -** *La finestra per aprire il File Chooser.*

Dopo aver dato l'OK, verrà visualizzata una finestra File Chooser, tramite la quale si dovrà localizzare la directory del JDK sul disco. Una volta trovata la cartella, non resta che premere il pulsante Locate JDK Directory.

# Parole chiave

ANDREA GINI

| | | | |
|---|---|---|---|
| abstract | double | int | super |
| boolean | else | interface | switch |
| break | extends | long | synchronized |
| byte | final | native | this |
| case | float | package | throw |
| char | for | private | throws |
| class | (goto) | protected | transient |
| (const) | if | public | try |
| continue | implements | return | void |
| default | import | short | volatile |
| do | instanceof | static | while |

La maggior delle parole riservate di Java deriva dal C, il linguaggio dal quale Java ha ereditato la sintassi delle strutture di controllo. Le somiglianze con il C++, al contrario, sono minori, dal momento che Java adotta una sintassi differente per quanto riguarda i costrutti caratteristici della programmazione a oggetti. Le parole chiave goto e const, presenti nel C, fanno parte dell'insieme delle keyword, ma di fatto non compaiono nel linguaggio Java: in questo modo, il compilatore può segnalare uno speciale messaggio di errore se il programmatore le dovesse utilizzare inavvertitamente.

Appendice D

# Diagrammi di classe e sistemi orientati agli oggetti

Andrea Gini

Un effetto della strategia di incapsulamento è quello di spingere il programmatore a esprimere il comportamento di un sistema a oggetti unicamente attraverso l'interfaccia di programmazione delle classi. In questo senso, quando un programmatore si trova a dover utilizzare una libreria di classi realizzate da qualcun altro, non è interessato a come essa sia stata effettivamente implementata: di norma, è sufficiente conoscere le firme dei metodi, le relazioni di parentela tra le classi, le associazioni e le dipendenze, informazioni che non dipendono dall'implementazione dei singoli metodi.

Il diagramma di classe è un formalismo che permette di rappresentare per via grafica tutte queste informazioni, nascondendo nel contempo i dettagli di livello inferiore. L'uso dei diagrammi di classe permette di vedere un insieme di classi Java da una prospettiva più alta rispetto a quella fornita dal codice sorgente, simile a quella che si ha quando si guarda una piantina per vedere com'è fatta una città. La piantina non contiene tutti i dettagli della zona rappresentata, come la posizione delle singole abitazioni o dei negozi, ma riporta informazioni sufficienti per orientarsi con precisione.

I diagrammi di classe fanno parte di UML (Unified Modeling Language), un insieme di notazioni grafiche che permette di fornire una rappresentazione dei diversi aspetti di un sistema software orientato agli oggetti, indipendentemente dal linguaggio di programmazione effettivamente utilizzato. L'UML comprende sette tipi diversi di diagrammi, che permettono di modellare i vari aspetti dell'architettura e del comportamento di un sistema software prima di iniziarne lo sviluppo. I diagrammi UML costituiscono una parte fondamentale della documentazione di un sistema informativo, e forniscono una guida essenziale in fase di studio o di manutenzione del sistema stesso.

L'UML non è un linguaggio di programmazione, anche se negli ultimi anni gli ambienti di sviluppo hanno iniziato a includere strumenti che permettono di produrre codice a partire dai

diagrammi e viceversa. I seguenti paragrafi vogliono fornire una guida essenziale ai diagrammi di classe, l'unico formalismo UML presente in questo libro.

# Classi e interfacce UML

In UML le classi e le interfacce sono rappresentate come rettangoli, suddivisi in tre aree: l'area superiore contiene il nome della classe o dell'interfaccia, quella intermedia l'elenco degli attributi e quella inferiore l'elenco dei metodi:

**Figura D.1 –** *Un esempio di classe e di interfaccia in UML.*

Entrambi i diagrammi non contengono alcun dettaglio sul contenuto dei metodi: il comportamento di una classe o di un'interfaccia UML è espresso unicamente tramite il nome dei suoi metodi. Le firme di metodi e attributi seguono una convenzione differente rispetto a quella adottata in Java: in questo caso, il nome precede il tipo, e tra i due compare un simbolo di due punti (:) come separatore. I parametri dei metodi, quando presenti, seguono la stessa convenzione. Il simbolo più (+), presente all'inizio, denota un modificatore public, mentre il trattino (-) indica private e il cancelletto (#) significa protected.

Il diagramma di interfaccia presenta alcune differenze rispetto a quello di classe:

- Al di sopra del nome compare un'etichetta "interface".
- Gli attributi (normalmente assenti) sono sottolineati, a indicare che si tratta di attributi statici immutabili.
- I metodi sono scritti in corsivo, per indicare che sono privi di implementazione.

Si osservi una tipica implementazione Java del diagramma di classe presente in figura 1:

```
public class SchedaAnagrafica {
```

```
    private String nome;
    private String cognome;

    public String getNome() {
      return nome;
    }
    public void setNome(String nome) {
          this.nome = nome;
    }
    public String getCognome() {
      return cognome;
    }
    public void setCognome(String cognome) {
      this.cognome = cognome;
    }
}
```

Spesso il diagramma di classe presenta un livello di dettaglio inferiore rispetto al codice sottostante: tipicamente, si usa un diagramma per descrivere un particolare aspetto di una classe, e si omettono i metodi e gli attributi che non concorrono a definire tale comportamento. In questo libro, i diagrammi di classe sono stati disegnati secondo questa convenzione.

# Ereditarietà e realizzazione

L'ereditarietà è rappresentata in UML con una freccia dalla punta triangolare, che parte dalla classe figlia e punta alla classe padre:

**Figura D.2 –** *Ereditarietà tra le classi.*

La realizzazione, equivalente all'implementazione di un'interfaccia in Java, viene rappresentata con una freccia simile a quella usata per l'ereditarietà, ma tratteggiata. Si noti che si ricorre alla realizzazione solo quando una classe implementa un'interfaccia, mentre se un'interfaccia ne estende un'altra si utilizza la normale ereditarietà.

In figura D.3 è possibile vedere un diagramma di classe contenente una relazione di ereditarietà tra interfacce (l'interfaccia MediaRecorder è figlia dell'interfaccia MediaPlayer) e due casi di realizzazione (la classe LettoreDVD realizza l'interfaccia MediaPlayer, mentre la classe VideoRegistratore realizza MediaRecorder).

**Figura D.3 –** *Un diagramma che contiene sia l'ereditarietà sia la realizzazione.*

# Associazione

L'associazione, rappresentata da una linea che congiunge due classi, denota una relazione di possesso. Un'associazione può essere bidirezionale o unidirezionale. Nel secondo caso, al posto di una linea semplice si utilizza una freccia. La freccia indica la direzione del flusso della comunicazione: in pratica, la classe da cui parte la freccia può chiamare i metodi di quella indicata dalla punta, ma non viceversa. L'equivalente Java dell'associazione è la presenza di un attributo in una classe, che di fatto denota il possesso di un particolare oggetto e la possibilità di invocare metodi su di esso.

**Figura D.4 –** *Classi unite da associazioni.*

In figura D.4 è possibile osservare un insieme di classi caratterizzate da associazioni sia uni- sia bidirezionali: un computer è collegato alle schede audio e video da associazioni bi direzionali, a indicare che la comunicazione avviene in entrambe le direzioni; le due schede, invece, presentano un'associazione unidirezionale rispettivamente con gli speaker e il monitor, poiché non è permessa la comunicazione in senso inverso.

# Aggregazione

Un tipo speciale di associazione è l'aggregazione, rappresentata da una linea tra due classi con un'estremità a diamante, che denota un'associazione uno a molti. In figura D.5 si può osservare una relazione uno a molti tra una biblioteca e i libri in essa contenuti.

**Figura D.5 –** *Un esempio di aggregazione.*

A parte la cardinalità, l'aggregazione equivale a un'associazione: nell'esempio di figura D.5 la classe Biblioteca possiede una collezione di libri e può invocare metodi su ognuno di essi. In Java, l'aggregazione corrisponde solitamente a un attributo di tipo `Vector` o `HashTable`, o più semplicemente a un array.

# Dipendenza

La dipendenza è rappresentata da una freccia tratteggiata. Letteralmente, la dipendenza suggerisce che la classe puntata dalla freccia esista indipendentemente dalla classe da cui parte la freccia: in Java, questo significa che la prima può essere compilata e utilizzata anche in assenza della seconda.

**Figura D.6 –** *Relazione di dipendenza in UML.*

La figura D.6 presenta un esempio di relazione di dipendenza: il CD, inteso come supporto per musica e dati, esiste indipendentemente dal particolare lettore con cui lo si legge: le sue caratteristiche sono definite da un documento denominato Red Book, al quale si devono attenere i produttori di lettori CD. Si noti l'etichetta "play" che compare sopra la freccia: le etichette permettono di fornire maggiori informazioni sul tipo di relazione che sussiste tra due classi.

# JavaBeans

Andrea Gini

## La programmazione a componenti

Uno degli obiettivi più ambiziosi dell'ingegneria del software è organizzare lo sviluppo di sistemi in maniera simile a quanto è stato fatto in altre branche dell'ingegneria, dove la presenza di un mercato di parti standard altamente riutilizzabili permette di aumentare la produttività riducendo nel contempo i costi. Nella meccanica, ad esempio, esiste da tempo un importante mercato di componenti riutilizzabili, come viti, dadi, bulloni e ruote dentate; ciascuno di questi componenti trova facilmente posto in centinaia di prodotti diversi.

L'industria del software, sempre più orientata alla filosofia dei componenti, sta dando vita a due nuove figure di programmatore: il progettista di componenti e l'assemblatore di applicazioni.

Il primo ha il compito di scoprire e progettare oggetti software di uso comune, che possano essere utilizzati con successo in contesti differenti. Produttori in concorrenza tra di loro possono realizzare componenti compatibili, ma con caratteristiche prestazionali differenti. L'acquirente può orientarsi su un mercato che offre una pluralità di scelte e decidere in base al budget o a particolari esigenze di prestazione.

L'assemblatore di applicazioni, d'altra parte, è un professionista specializzato in un particolare dominio applicativo, capace di creare programmi complessi acquistando sul mercato componenti standard e combinandoli con strumenti grafici o linguaggi di scripting.

Questo capitolo offre un'analisi approfondita delle problematiche che si incontrano nella creazione di componenti in Java; attraverso gli esempi verrà comunque offerta una panoramica su come sia possibile assemblare applicazioni complesse a partire da componenti concepiti per il riuso.

# La specifica JavaBeans

JavaBeans è una specifica, ossia un insieme di regole seguendo le quali è possibile realizzare in Java componenti software riutilizzabili, che abbiano la capacità di interagire con altri componenti, realizzati da altri produttori, attraverso un protocollo di comunicazione comune.

Ogni Bean è caratterizzato dai servizi che è in grado di offrire e può essere utilizzato in un ambiente di sviluppo differente rispetto a quello in cui è stato realizzato. Quest'ultimo punto è cruciale nella filosofia dei componenti: sebbene i Java Beans siano a tutti gli effetti classi Java, e possano essere manipolati completamente per via programmatica, essi vengono spesso utilizzati in ambienti di sviluppo diversi, come tool grafici o linguaggi di scripting.

I tool grafici, tipo JBuilder, permettono di manipolare i componenti in maniera visuale. Un assemblatore di componenti può selezionare i Beans da una palette, inserirli in un apposito contenitore, impostarne le proprietà, collegare gli eventi di un Bean ai metodi di un altro, generando in tal modo applicazioni, Applet, Servlet e persino nuovi componenti senza scrivere una sola riga di codice.

I linguaggi di scripting, di contro, offrono una maggiore flessibilità rispetto ai tool grafici, senza presentare le complicazioni di un linguaggio generico. La programmazione di pagine web dinamiche, uno dei domini applicativi di maggior attualità, deve il suo rapido sviluppo a un'intelligente politica di stratificazione, che vede le funzionalità di più basso livello, come la gestione dei database, la Businnes Logic o l'interfacciamento con le risorse di sistema, incapsulate all'interno di JavaBeans, mentre tutto l'aspetto della presentazione viene sviluppato con un semplice linguaggio di scripting, tipo Java Server Pages o PHP.

# Il modello a componenti JavaBeans

Un modello a componenti è caratterizzato da almeno sette fattori: proprietà, metodi, introspezione, personalizzazione, persistenza, eventi e modalità di deployment. Nei prossimi paragrafi si analizzerà il ruolo di ciascuno di questi aspetti all'interno della specifica Java Beans; quindi si procederà a descriverne l'implementazione in Java.

## Proprietà

Le proprietà sono attributi privati, accessibili solamente attraverso appositi metodi `get` e `set`. Tali metodi costituiscono l'unica via di accesso pubblica alle proprietà, cosa che permette al progettista di componenti di stabilire per ogni parametro precise regole di accesso. Se si utilizzano i Bean all'interno di un programma di sviluppo visuale, le proprietà di un componente vengono visualizzate in un apposito pannello, che permette di modificarne il valore con un opportuno strumento grafico.

## Metodi

I metodi di un Bean sono metodi pubblici Java, con l'unica differenza che essi risultano accessibili anche attraverso linguaggi di scripting e Builder Tools. I metodi sono la prima e più importante via d'accesso ai servizi di un Bean.

## Introspezione

I Builder Tools scoprono i servizi di un Bean (proprietà, metodi ed eventi) attraverso un processo noto come introspezione, che consiste principalmente nell'interrogare il componente per conoscerne i metodi, e dedurre da questi le caratteristiche. Il progettista di componenti può attivare l'introspezione in due maniere: seguendo precise convenzioni nella formulazione delle firme dei metodi, o creando una speciale classe `BeanInfo`, che fornisce un elenco esplicito dei servizi di un particolare Bean.

La prima via è senza dubbio la più semplice: se si definiscono i metodi di accesso a un determinato servizio seguendo le regole di naming descritte dalla specifica JavaBeans, i tool grafici saranno in grado, grazie alla reflection, di individuare i servizi di un Bean semplicemente osservandone l'interfaccia di programmazione. Il ricorso ai BeanInfo, d'altro canto, torna utile in tutti quei casi in cui sia necessario mascherare alcuni metodi, in modo da esporre solamente un sottoinsieme dei servizi effettivi del Bean.

## Personalizzazione

Durante il lavoro di composizione di Java Beans all'interno di un tool grafico, un apposito Property Sheet, generato al volo dal programma di composizione, mostra lo stato delle proprietà e permette di modificarle con un opportuno strumento grafico, tipo un campo di testo per valori `String` o una palette per proprietà `Color`. Simili strumenti grafici vengono detti editor di proprietà.

I tool grafici dispongono di editor di proprietà in grado di supportare i tipi Java più comuni, come i tipi numerici, le stringhe e i colori; nel caso si desideri rendere editabile una proprietà di un tipo diverso, è necessario realizzare un'opportuna classe di supporto, conforme all'interfaccia `PropertyEditor`. Quando invece si desideri fornire un controllo totale sulla configurazione di un Bean, è possibile definire un Bean Customizer, una speciale applicazione grafica specializzata nella configurazione di un particolare tipo di componenti.

## Persistenza

La persistenza permette ad un Bean di salvare il proprio stato e di ripristinarlo in un secondo tempo. JavaBeans supporta la persistenza grazie all'Object Serialization, che permette di risolvere questo problema in modo molto rapido.

## Eventi

Nella programmazione a oggetti tradizionale non esiste nessuna convenzione su come modellare lo scambio di messaggi tra oggetti. Ogni programmatore adotta un proprio sistema, creando una fitta rete di dipendenze che rende molto difficile il riutilizzo di oggetti in contesti differenti da quello di partenza. Gli oggetti Java progettati secondo la specifica Java Beans adottano un meccanismo di comunicazione basato sugli eventi, simile a quello utilizzato nei componenti grafici Swing e AWT. L'esistenza di un unico protocollo di comunicazione standard garantisce l'intercomunicabilità tra componenti, indipendentemente da chi li abbia prodotti.

## Deployment

I JavaBeans possono essere consegnati, in gruppo o singolarmente, attraverso file JAR, speciali archivi compressi in grado di trasportare tutto quello di cui un Bean ha bisogno, come classi, immagini o altri file di supporto. Grazie ai file .jar è possibile consegnare i Beans con una modalità del tipo "chiavi in mano": l'acquirente deve solamente caricare un file JAR nel proprio ambiente di sviluppo e i Beans in esso contenuti verranno subito messi a disposizione. L'impacchettamento di classi Java all'interno di file JAR segue poche semplici regole, che verranno descritte negli esempi del capitolo.

# Guida all'implementazione dei JavaBeans

Realizzare un componente Java Bean è un compito alla portata di qualunque programmatore Java che disponga di buone conoscenze di sviluppo Object Oriented. Nei paragrafi seguenti verranno descritte dettagliatamente le convenzioni di naming dettate dalla specifica, e verranno fornite le istruzioni su come scrivere le poche righe di codice necessarie a implementare i meccanismi che caratterizzano i servizi Bean. Infine verranno presentati degli esempi, che permetteranno di impratichirsi con il processo di implementazione delle specifiche.

## Le proprietà

Le proprietà sono attributi che descrivono l'aspetto e il comportamento di un Bean, e che possono essere modificate durante tutto il ciclo di vita del componente. Di base, le proprietà sono attributi privati, ai quali si accede attraverso una coppia di metodi della forma:

```
public <PropertyType> get<PropertyName>()
public void set<PropertyName>(<PropertyType> property)
```

La convenzione di aggiungere il prefisso get e set ai metodi che forniscono l'accesso a una proprietà, permette ad esempio ai tool grafici di rilevare le proprietà Bean, determinarne le regole di accesso (Read Only o Read/Write), dedurne il tipo, visualizzare le proprietà su un apposito Property Sheet e individuare l'editor di proprietà più adatto al caso.
Se ad esempio un tool grafico scopre, grazie all'introspezione, la coppia di metodi

```
public Color getForegroundColor() { ... }
public void setForegroundColor(Color c) { ... }
```

da questi conclude che esiste una proprietà chiamata foregroundColor (notare la prima lettera minuscola), accessibile sia in lettura che in scrittura, di tipo Color. A questo punto, il tool può cercare un editor di proprietà per parametri di tipo Color, e mostrare la proprietà su un property sheet in modo che possa essere vista e manipolata dal programmatore.

## Proprietà indicizzate (Indexed Property)

Le proprietà indicizzate permettono di gestire collezioni di valori accessibili attraverso indice, in maniera simile a come si fa con un vettore. Lo schema di composizione dei metodi di accesso di una proprietà indicizzata è il seguente:

```
public <PropertyType>[] get<PropertyName>();
public void set<PropertyName>(<PropertyType>[] value);
```

per i metodi che permettono di manipolare l'intera collection, mentre per accedere ai singoli elementi, si deve predisporre una coppia di metodi del tipo:

```
public <PropertyType> get<PropertyName>(int index);
public void set<PropertyName>(int index, <PropertyType> value);
```

## Proprietà bound

Le proprietà semplici, così come sono state descritte nei paragrafi precedenti, seguono una convenzione radicata da tempo nella normale programmazione a oggetti. Le proprietà bound, al contrario, sono caratteristiche dell'universo dei componenti, dove si verifica molto spesso la necessità di collegare il valore delle proprietà di un componente a quelli di un'altro, in modo tale che si mantengano aggiornati. I metodi set delle proprietà bound, inviano una notifica a tutti gli ascoltatori registrati ogni qualvolta viene alterato il valore della proprietà. Il meccanismo di ascolto–notifica, simile a quello degli eventi Swing e AWT, segue il pattern Observer.

Le proprietà bound, a differenza degli eventi Swing, utilizzano un unico tipo di evento, ChangeEvent, cosa che semplifica il processo di implementazione. La classe PropertyChangeSupport, presente all'interno del package java.bean, fornisce i metodi che gestiscono la lista degli ascoltatori e quelli che producono l'invio degli eventi.

Un oggetto che voglia mettersi in ascolto di una proprietà, deve implementare l'interfaccia PropertyChangeListener e deve registrarsi presso la sorgente di eventi. L'oggetto PropertyChangeEvent incapsula le informazioni riguardo alla proprietà modificata, alla sorgente e al valore della proprietà.

**Figura 18.1 –** *Il meccanismo di notifica di eventi bound segue il pattern Observer*

## Come implementare il supporto alle proprietà bound

Per aggiungere a un Bean il supporto alle proprietà bound, bisogna anzitutto importare il package java.beans.*, in modo da garantire l'accesso alle classi PropertyChange-Support e PropertyChangeEvent. Quindi bisogna creare un oggetto PropertyChangeSupport, che ha il compito di mantenere la lista degli ascoltatori e di fornire i metodi che gestiscono l'invio degli eventi.

```
private PropertyChangeSupport changes = new PropertyChangeSupport(this);
```

A questo punto bisogna realizzare, nella propria classe, i metodi che permettono di gestire la lista degli ascoltatori. Tali metodi sono dei semplici metodi Wrapper che fanno riferimento a metodi con la stessa firma, presenti nel PropertyChangeSupport:

```
public void addPropertyChangeListener(PropertyChangeListener l) {
 changes.addPropertyChangeListener(l);
}
```

```
public void removePropertyChangeListener(PropertyChangeListener l) {
 changes.removePropertyChangeListener(l);
}
```

La presenza dei metodi addPropertyChangeListener() e removeProperty-ChangeListener() permette ai tool grafici di riconoscere un oggetto in grado di inviare proprietà bound e di mettere a disposizione un'opportuna voce nel menù di gestione degli eventi.

L'ultimo passaggio consiste nel modificare i metodi set relativi alle proprietà che si vuole rendere bound, per fare in modo che venga generato un PropertyChangeEvent ogni volta che la proprietà viene reimpostata

```
public void setColor(Color newColor) {
 Color oldColor = color;
 color = newColor;
 changes.firePropertyChange("color", oldColor , newColor);
}
```

Nel caso di proprietà read only, prive di metodo set, l'invio dell'evento dovrà avvenire all'interno del metodo che attua la modifica della proprietà. Un aspetto interessante del meccanismo di invio di PropertyChangeEvent, è che essi trasportano sia il nuovo valore che quello vecchio. Questa scelta dispensa chi implementa un ascoltatore dal compito di mantenere una copia del valore, qualora questo fosse necessario, dal momento che l'evento viene propagato *dopo* la modifica della relativa proprietà. Il metodo fireChangeEvent() della classe PropertyChangeListener fornisce il servizio di Event Dispatching:

```
firePropertyChange(String propertyName, Object oldValue, Object newValue)
```

In pratica esso impacchetta i parametri in un oggetto PropertyChangeEvent, e chiama il metodo propertyChange(PropertyChangeEvent p) su tutti gli ascoltatori registrati. I parametri vengono trattati come Object, e nel caso si debbano inviare proprietà espresse in termini di tipi primitivi, occorre incapsularle nell'opportuno Wrapper (Integer per valori int, Double per valori double e così via). Per facilitare questo compito, la classe propertyChangeSupport prevede delle varianti di firePropertyChange per valori int e boolean.

## Come implementare il supporto alle proprietà bound su sottoclassi di JComponent

La classe JComponent, superclasse di tutti i componenti Swing, dispone del supporto nativo alla gestione di proprietà bound. Di base essa fornisce i metodi addProperty-ChangeListener e removePropertyChangeListener, oltre a una collezione di metodi firePropertyChange adatta ad ogni tipo primitivo. In questo caso l'implementazione di una proprietà bound richiederà solo una modifica al metodo set preposto, similmente a come descritto nell'ultimo passaggio del precedente paragrafo, con la differenza che non è necessario ricorrere a un oggetto propertyChangeSupport per inviare la proprietà:

```
public void setColor(Color newColor) {
 Color oldColor = color;
 color = newColor;
 firePropertyChange("color", oldColor , newColor);
}
```

### Ascoltatori di proprietà

Se si desidera mettersi in ascolto di una proprietà, occorre definire un opportuno oggetto PropertyChangeListener e registrarlo presso il Bean. Un Property-ChangeListener deve definire il metodo propertyChange(PropertyChangeEvent e), che viene chiamato quando avviene la modifica di una proprietà bound.

Un PropertyChangeListener viene notificato quando avviene la modifica di *una qualunque* proprietà bound: per questa ragione esso deve, come prima cosa, verificare, che la proprietà appena modificata sia quella alla quale si è interessati. Una simile verifica richiede una chiamata al metodo getPropertyName di PropertyChangeEvent, che restituisce il nome della proprietà. Per convenzione, i nomi di proprietà vengono estratti dai nomi dichiarati nei metodi get e set, con la prima lettera minuscola. Il seguente frammento di codice presenta un tipico PropertyChange-Listener, che ascolta la proprietà foregroundColor:

```
public class Listener implements PropertyChangeListener() {
 public void propertyChange(PropertyChangeEvent e) {
   if(e.getPropertyName().equals("foregroundColor"))
     System.out.println(e.getNewValue());
 }
}
```

## Un esempio di Bean con proprietà bound

Un Java Bean rappresenta un mattone di un programma. Ogni componente è un'unità di utilizzo abbastanza grossa da incorporare una funzionalità evoluta, ma piccola rispetto ad un programma fatto e finito. Il concetto del riuso può essere presente a diversi livelli del progetto: il seguente Bean fornisce un esempio di elevata versatilità

Il Bean PhotoAlbum è un pannello grafico al cui interno vengono caricate delle immagini. Il metodo showNext() permette di passare da un'immagine all'altra, in modo ciclico. Il numero ed il tipo di immagini viene determinato al momento dell'avvio: durante la fase di costruzione viene letto il file comment.txt, presente nella directory images, che contiene una riga di commento per ogni immagine presente nella cartella. Le immagini devono essere nominate in modo progressivo (img0.jpg, img1.jpg, img2.jpg…) e devono essere presenti in numero uguale alle righe del file comment.txt. Questa scelta progettuale consente di introdurre il riuso a un livello abbastanza alto: qualunque utente, anche con scarse conoscenze del linguaggio, può personalizzare il componente, inserendo le sue foto preferite, senza la necessità di alterare il codice sorgente.

Il Bean PhotoAlbum ha tre proprietà:

- imageNumber, che restituisce il numero di immagini contenute nell'album. Essendo una quantità immutabile, tale proprietà è stata implementata come proprietà semplice.

- imageIndex: restituisce l'indice dell'immagine attualmente visualizzata. Al cambio di immagine viene inviato un PropertyChangeEvent.

- imageComment: restituisce una stringa di commento all'immagine. Anche in questo caso, al cambio di immagine viene generato un PropertyChangeEvent.

Il Bean viene definito come sottoclasse di JPanel: per questo motivo non vengono dichiarati i metodi addPropertyChangeListener e removeProperty-ChangeListener, già presenti nella superclasse. L'invio delle proprietà verrà messo in atto grazie al metodo firePropertyChange di JComponent.

```
package com.mokabyte.mokabook.javaBeans.photoAlbum;

import java.awt.*;
import java.beans.*;
import java.io.*;
import java.net.*;
import java.util.*;
import javax.swing.*;

public class PhotoAlbum extends JPanel {

 private Vector comments = new Vector();
 private int imageIndex;

 public PhotoAlbum() {
   super();
   setLayout(new BorderLayout());
   setupComments();
   imageIndex = 0;
   showNext();
 }
 private void setupComments() {
   try {
     URL indexUrl = getClass().getResource("images/" + "comments.txt");
     InputStream in = indexUrl.openStream();
     BufferedReader lineReader = new BufferedReader(new InputStreamReader(in));
     String line;
     while((line = lineReader.readLine())!=null)
       comments.add(line);
   }
   catch(Exception e) {
     e.printStackTrace();
```

```
    }
  }
  public int getImageNumber() {
    return comments.size();
  }
  public int getImageIndex() {
    return imageIndex;
  }
  public String getImageComment() {
    return (String)comments.elementAt(imageIndex);
  }
  public void showNext() {
    int oldImageIndex = imageIndex;
    imageIndex = ((imageIndex +1 ) % comments.size());
    String imageName = «img» + Integer.toString(imageIndex) + «.jpg»;
    showImage(getClass().getResource(„images/“ + imageName));
    String oldImageComment = (String)comments.elementAt(oldImageIndex);
    String currentImageComment = (String)comments.elementAt(imageIndex);
    firePropertyChange(“imageComment”, oldImageComment, currentImageComment);
    firePropertyChange(“imageIndex”, oldImageIndex, imageIndex);
  }
  private void showImage(URL imageUrl) {
    ImageIcon img = new ImageIcon(imageUrl);
    JLabel picture = new JLabel(img);
    JScrollPane pictureScrollPane = new JScrollPane(picture);
    removeAll();
    add(BorderLayout.CENTER,pictureScrollPane);
    validate();
  }
}
```

È possibile testare il Bean come fosse una normale classe Java, utilizzando queste semplici righe di codice:

```
package com.mokabyte.mokabook.javaBeans.photoAlbum;

import com.mokabyte.mokabook.javaBeans.photoAlbum.*;
import java.beans.*;
import javax.swing.*;

public class PhotoAlbumTest {
public static void main(String argv[]) {
  JFrame f = new JFrame(“Photo Album”);
  PhotoAlbum p = new PhotoAlbum();
  f.getContentPane().add(p);
  p.addPropertyChangeListener(new PropertyChangeListener() {
    public void propertyChange(PropertyChangeEvent e) {
```

```
            System.out.println(e.getPropertyName() + ": " + e.getNewValue());
        }
      });
   f.setSize(500,400);
   f.setVisible(true);

   while(true)
      for(int i=0;i<7;i++) {
            p.showNext();
            try {Thread.sleep(1000);}catch(Exception e) {}
        }
   }
   }
```

**Figura 18.2 –** *Un programma di prova per il Bean PhotoAlbum*

## Creazione di un file JAR

Prima di procedere alla consegna del Bean entro un file JAR, bisogna anzitutto compilare le classi PhotoAlbum.java e PhotoAlbumTest.java, che devono trovarsi nella cartella com\mokabyte\mokabook\javaBeans\

```
javac com\mokabyte\mokabook\javaBeans\photoAlbum\*.java
```

A questo punto bisogna creare, ricorrendo a un semplice editor di testo tipo Notepad, un file photoAlbumManifest.tmp con il seguente contenuto

```
Main-Class: com.mokabyte.mokabook.javaBeans.photoAlbum.PhotoAlbumTest
```

```
Name: com/mokabyte/mokabook/javaBeans/photoAlbum/PhotoAlbum.class
Java-Bean: True
```

Le prime due righe, opzionali, segnalano la presenza di una classe dotata di metodo main.

Le ultime due righe del file manifest specificano che la classe PhotoAlbum.class è un Java Bean. Se l'archivio contiene più di un Bean, è necessario elencarli tutti.

Per generare l'archivio photoAlbum.jar, bisogna digitare la riga di comando:

```
jar cfm photoAlbum.jar photoAlbumManifest.tmp
 com\mokabyte\mokabook\javaBeans\photoAlbum\*.class
 com\mokabyte\mokabook\javaBeans\photoAlbum\images\*.*
```

Il file così generato contiene tutte le classi e le immagini necessarie a dar vita al Bean PhotoAlbum. Tale file potrà essere utilizzato facilmente all'interno di tool grafici o di pagine web, racchiuso dentro una Applet.

Il file .jar potrà essere avviato digitando

```
java PhotoAlbum.jar
```

**Figura 18.3** – *Un file JAR opportunamente confezionato può essere aperto con un opportuno tool come Jar o WinZip*

Le istruzioni fornite sono valide per la piattaforma Windows. Su piattaforma Unix, le eventuali occorrenze del simbolo "\", che funge da path separator su piattaforme Windows, andranno sostituite col simbolo "/". Le convenzioni adottate all'interno del file manifest valgono invece su entrambe le piattaforme.

### Integrazione con altri Bean

Nonostante il Bean PhotoAlbum fornisca un servizio abbastanza evoluto, non è ancora classificabile come applicazione. Esso, opportunamente integrato con altri Beans, può comunque dar vita a numerosi programmi; di seguito, ecco qualche esempio: collegato a un CalendarBean, PhotoAlbum può dar vita a un simpatico calendario elettronico; collegando un bottone Bean al metodo showNext() è possibile creare un album interattivo, impacchettarlo su un'Applet e pubblicarlo su Internet; impacchettando il Bean PhotoAlbum con foto natalizie, e collegandolo con un Bean Carillon, si può ottenere un biglietto di auguri elettronico.

**Figura 18.4** – *Combinando, all'interno del Bean Box, il Bean PhotoAlbum con un pulsante Bean, si ottiene una piccola applicazione*

A questi esempi se ne possono facilmente aggiungere altri; altri ancora diventano possibili aggiungendo al Bean nuovi metodi, come previousImage() e setImageAt(int i); un compito ormai alla portata del lettore che fornisce un ottimo pretesto per esercitarsi.

## Eventi Bean

La notifica del cambiamento di valore delle proprietà bound è un meccanismo di comunicazione tra Beans. Se si vuole che un Bean sia in grado di propagare eventi di tipo più generico, o comunque eventi che non è comodo rappresentare come un cambiamento di stato, è possibile utilizzare un meccanismo di eventi generico, del tutto simile a quello pre-

sente nei componenti grafici Swing e AWT. I prossimi paragrafi servono a illustrare le tre fasi dell'implementazione: creazione dell'evento, definizione dell'ascoltatore e infine creazione della sorgente di eventi.

### Creazione di un evento

Per implementare un meccanismo di comunicazione basato su eventi, occorre anzitutto definire un'opportuna sottoclasse di EventObject, che racchiuda tutte le informazioni relative all'evento da propagare.

```
public class <EventType> extends EventObject {
 private <ParamType> param
 public <EventType>(Object source,<ParamType> param) {
   super(source);
   this.param = param;
 }
 public <ParamType> getParameter() {
   return param;
 }
}
```

La principale variazione sul tema si ha sul numero e sul tipo di parametri: tanto più complesso è l'evento da descrivere, maggiori saranno i parametri in gioco. L'unico parametro che è obbligatorio fornire è un reference all'oggetto che ha generato l'evento: tale reference, richiamabile con il metodo getSource() della classe EventObject, permetterà all'ascoltatore di interrogare la sorgente degli eventi qualora ce ne fosse bisogno.

### Destinatari di eventi

Il secondo passaggio è quello di definire l'interfaccia di programmazione degli ascoltatori di eventi. Tale interfaccia deve essere definita come sottoclasse di EventListener, per essere riconoscibile come ascoltatore dall'Introspector. Lo schema di sviluppo degli ascoltatori segue lo schema

```
import java.awt.event.*;

public Interface <EventListener> extends EventListener {
 public void <EventType>Performed(<EventType> e);
}
```

Le convenzioni di naming dei metodi dell'interfaccia non seguono uno schema standard: la convenzione descritta nell'esempio, <EventType>performed, può essere seguita o meno. L'importante è che il nome dei metodi dell'interfaccia Listener suggeriscano il tipo di azione sottostante, e che accettino come parametro un evento del tipo giusto.

## Sorgenti di eventi

Se si desidera aggiungere a un Bean la capacità di generare eventi, occorre implementare una coppia di metodi

```
add<EventListenerType>(<EventListenerType> l)
remove<EventListenerType>(<EventListenerType> l)
```

La gestione della lista degli ascoltatori e l'invio degli eventi segue una formula standard, descritta nelle righe seguenti:

```
private Vector listeners = new Vector();

public void add<EventListenerType>(<EventListenerType> l) {
 listeners.add(l);
}
public void remove<EventListenerType>(<EventListenerType> l) {
 listeners.remove(l);
}
protected void fire<Eventype>(<EvenType> e) {
 Enumeration listenersEnumeration = listeners.elements();
 while(listenersEnumeration.hasMoreElements()) {
 <EventListenerType> listener = (<EventListenerType>)listenersEnumeration.nextElement();
   listener.<EventType>Performed(e);
 }
}
```

## Sorgenti unicast

In alcuni casi occorre definire sorgenti di eventi capaci di servire un unico ascoltatore. Per implementare tali classi, che fungono da sorgenti unicast, si può seguire il seguente modello

```
private <EventListenerType> listener;

public void add<EventListenerType>(
<EventListenerType> l) throws TooManyListenersException {
 if(listener == null)
   listener = l;
 else
   throw new TooManyListenerException();
}
public void remove<EventListenerType>(<EventListenerType> l) {
 listener = null;
}
protected void fire<Eventype>(<EvenType> e) {
```

```
if(listener! = null)
    listener.<EventType>Performed(e);
}
```

## Ascoltatori di eventi: Event Adapter

Se si vuole che un evento generato da un Bean scateni un'azione su un altro Bean, è necessario creare un oggetto che realizzi un collegamento tra i due. Tale classe, detta Adapter, viene registrata come ascoltatore presso la sorgente dell'evento, e formula una chiamata al metodo destinazione ogni volta che riceve una notifica dal Bean sorgente.

Gli strumenti grafici tipo JBuilder generano questo tipo di classi in maniera automatica: tutto quello che l'utente deve fare è collegare, con pochi click di mouse, l'evento di un Bean sorgente a un metodo di un Bean target. Qui di seguito viene riportato il codice di un Adapter, generato automaticamente dal Bean Box, che collega la pressione di un pulsante al metodo `startJuggling(ActionEvent e)` del Bean `Juggler`.

```
// Automatically generated event hookup file.

public class ___Hookup_172935aa26 implements    java.awt.event.ActionListener, java.io.Serializable {

 public void setTarget(sunw.demo.juggler.Juggler t) {
    target = t;
 }

 public void actionPerformed(java.awt.event.ActionEvent arg0) {
    target.startJuggling(arg0);
 }

 private sunw.demo.juggler.Juggler target;
}
```

**Figura 18.5 –** *Un Adapter funge da ponte di collegamento tra gli eventi di un Bean e i metodi di un altro*

**EventSource**

+addEventListener()
+removeEventListener()
+fireEvent()

## Un esempio di Bean con eventi

Il prossimo esempio è un Bean Timer, che ha il compito di generare battiti di orologio a intervalli regolari. Questo componente è un tipico esempio di Bean non grafico.

La prima classe che si definisce è quella che implementa il tipo di evento

```
package com.mokabyte.mokabook.javaBeans.timer;

import com.mokabyte.mokabook.javaBeans.*;
import java.io.*;
import java.util.*;

public class TimerEvent extends EventObject implements Serializable {

 public TimerEvent(Object source) {
    super(source);
 }
}
```

Come si può vedere, l'implementazione di un nuovo tipo di evento è questione di poche righe di codice. L'unico particolare degno di nota è che il costruttore del nuovo tipo di evento deve invocare il costruttore della superclasse, passando un reference alla sorgente dell'evento.

L'interfaccia che rappresenta l'ascoltatore deve estendere l'interfaccia EventListener; a parte questo, al suo interno si può definire un numero arbitrario di metodi, la cui unica costante è quella di avere come parametro un reference all'evento da propagare.

```
package com.mokabyte.mokabook.javaBeans.timer;

import com.mokabyte.mokabook.javaBeans.timer.*;

public interface TimerListener extends java.util.EventListener {
 public void clockTicked(TimerEvent e);
}
```

Per finire, ecco il Bean vero e proprio. Come si può notare, esso implementa l'interfaccia Serializable che rende possibile la serializzazione.

```
package com.mokabyte.mokabook.javaBeans.timer;

import com.mokabyte.mokabook.javaBeans.*;
import java.io.*;
import java.util.*;

public class TimerBean implements Serializable {
```

```
private int time = 1000;
private transient TimerThread timerThread;
private Vector timerListeners = new Vector();

public void addTimerListener(TimerListener t) {
 timerListeners.add(t);
}
public void removeTimerListener(TimerListener t) {
 timerListeners.remove(t);
}
protected void fireTimerEvent(TimerEvent e) {
 Enumeration listeners = timerListeners.elements();
 while(listeners.hasMoreElements())
    ((TimerListener)listeners.nextElement()).clockTicked(e);
}
public synchronized void setMillis(int millis) {
 time = millis;
}
public synchronized int getMillis() {
 return time;
}
public synchronized void startTimer() {
 if(timerThread!=null)
    forceTick();
 timerThread = new TimerThread();
 timerThread.start();
}
public synchronized void stopTimer() {
 if(timerThread == null)
    return;

 timerThread.killTimer();
 timerThread = null;
}
public synchronized void forceTick() {
 if(timerThread!=null) {
    stopTimer();
    startTimer();
 }
 else
    fireTimerEvent(new TimerEvent(this));
}

 class TimerThread extends Thread {
    private boolean running = true;

    public synchronized void killTimer() {
```

```
            running = false;
        }
        private synchronized boolean isRunning() {
            return running;
        }
        public void run() {
            while(true)
                try {
                    if(isRunning()) {
                        fireTimerEvent(new TimerEvent(TimerBean.this));
                        Thread.sleep(getMillis());
                    }
                    else
                        break;
            catch(InterruptedException e) {}
            }
        }
    }
}
```

I primi tre metodi servono a gestire la lista degli ascoltatori. Il terzo e il quarto gestiscono la proprietà millis, ossia il tempo, in millisecondi, tra un tick e l'altro. I due metodi successivi, startTimer, stopTimer, servono ad avviare e fermare il timer, mentre forceTick lancia un tick e riavvia il timer, se questo è attivo. Il timer vero e proprio viene implementato grazie a una classe interna TimerThread, sottoclasse di Thread. Si noti il metodo killTimer, che permette di terminare in modo pulito la vita del thread: questa soluzione è da preferire al metodo stop (deprecato a partire dal JDK 1.1), che in certi casi può provocare la terminazione del thread in uno stato inconsistente.

Per compilare le classi del Bean, bisogna usare la seguente riga di comando

```
javac com\mokabyte\mokabook\javaBeans\timer\*.java
```

Per impacchettare il Bean in un file .jar, è necessario per prima cosa creare con un editor di testo il file timerManifest.tmp, con le seguenti righe

```
Name: com/mokabyte/mokabook/javaBeans/timer/TimerBean.class
Java-Bean: True
```

Per creare l'archivio si deve quindi digitare il seguente comando

```
jar cfm timer.jar timerManifest.tmp com\mokabyte\mokabook\javaBeans\timer\*.class
```

Per testare la classe TimerBean, si può usare il seguente programma, che crea un oggetto TimerBean e registra un TimerListener il quale stampa a video una scritta ad ogni tick del timer.

```
package com.mokabyte.mokabook.javaBeans.timer;

import com.mokabyte.mokabook.javaBeans.timer.*;

public class TimerTest {
 public static void main(String argv[]) {


TimerBean t = new TimerBean();
   t.addTimerListener(new TimerListener() {
      public void clockTicked(TimerEvent e) {
         System.out.println("Tick");
      }
   });
   t.startTimer();
 }
}
```

# Introspezione: l’interfaccia BeanInfo

Le convenzioni di naming descritte nei paragrafi precedenti permettono ai tool grafici abilitati ai Beans di scoprire i servizi di un componente grazie alla reflection. Questo processo automatico è certamente comodo, ma ha il difetto di non offrire nessun tipo di controllo sul numero e sul tipo di servizi da mostrare. In alcune occasioni può essere necessario mascherare un certo numero di servizi, specie quelli ereditati da una superclasse.

I Beans creati a partire dalla classe JComponent, ad esempio, ereditano automaticamente più di dieci attributi (dimensioni, colore, allineamento…) e ben dodici tipi diversi di evento (ComponentEvent, MouseEvent, HierarchyEvent…). Un simile eccesso provoca di solito disorientamento nell’utente; in questi casi è preferibile fornire un elenco esplicito dei servizi da associare al nostro Bean, in modo da “ripulire” gli eccessi.

Per raggiungere questo obiettivo, bisogna associare al Bean una classe di supporto, che implementi l’interfaccia BeanInfo. Una classe BeanInfo permette di fare un certo numero di cose: esporre solamente i servizi che si desidera rendere visibili, aggirare le convenzioni di naming imposte dalle specifiche Java Beans, associare al Bean un’icona e attribuire ai servizi nomi più descrittivi di quelli rilevabili con il processo di analisi delle firme dei metodi.

## Creare una classe BeanInfo

Per creare una classe BeanInfo bisogna anzitutto definire una classe con lo stesso nome del Bean, a cui si deve aggiungere il suffisso BeanInfo. Per semplificare il lavoro si può estendere SimpleBeanInfo, una classe che fornisce un’implementazione nulla di tutti i metodi dell’interfaccia. In questo modo ci si limiterà a sovrascrivere solamente i metodi che interessano, lasciando tutti gli altri con l’impostazione di default.

Per ridefinire il numero ed il tipo dei servizi Bean, occorre agire in modo appropriato a restituire le proprietà, i metodi o gli eventi che si desidera esporre. Opzionalmente, si può associare

un'icona al Bean, definendo il metodo public java.awt.Image getIcon(int iconKind). Per finire, si può specificare la classe del Bean e il suo Customizer, qualora ne esista uno, con il metodo public BeanDescriptor getBeanDescriptor().

La classe BeanInfo così prodotta deve essere messa nello stesso package che contiene il Bean. In assenza di una classe BeanInfo, i servizi di un Bean vengono trovati con la reflection.

### Feature Descriptors

Una classe di tipo BeanInfo restituisce, tramite i seguenti metodi, vettori di *descriptors* che contengono informazioni relative ad ogni proprietà, metodo o evento che il progettista di un Bean desidera esporre.

```
PropertyDescriptor[] getPropertyDescriptors();
MethodDescriptor[] getMethodDescriptors();
EventSetDescriptor[] getEventSetDescriptors();
```

Ogni Descriptor fornisce una precisa rappresentazione di una classe di servizi Bean. Il package java.bean implementa le seguenti classi:

- FeatureDescriptor: è la classe base per tutte le altre classi Descriptor, e definisce gli aspetti comuni a tutta la famiglia.
- BeanDescriptor: descrive il tipo e il nome della classe Bean associati, oltre a fornire il Customizer, se ne esiste uno.
- PropertyDescriptor: descrive le proprietà del Bean.
- IndexedPropertyDescriptor: è una sottoclasse di PropertyDescriptor, e descrive le proprietà indicizzate.
- EventSetDescriptor: descrive gli eventi che il Bean è in grado di inviare.
- MethodDescriptor: descrive i metodi del Bean.
- ParameterDescriptor: descrive i parametri dei metodi.

## Esempio

In questo esempio si analizzerà un BeanInfo per il Bean PhotoAlbum, che permette di nascondere una grossa quantità di servizi Bean che per default vengono ereditati dalla superclasse JPanel.

```
package com.mokabyte.mokabook.javaBeans.photoAlbum;

import java.beans.*;
```

```
import com.mokabyte.mokabook.javaBeans.photoAlbum.*;

public class PhotoAlbumBeanInfo extends SimpleBeanInfo {

private static final Class beanClass = PhotoAlbum.class;

 public PropertyDescriptor[] getPropertyDescriptors() {
   try {
      PropertyDescriptor imageNumber
      = new PropertyDescriptor("imageNumber", beanClass, "getImageNumber", null);
      PropertyDescriptor imageIndex = new PropertyDescriptor("imageIndex", beanClass, "getImageIndex", null);
      PropertyDescriptor imageComment
      = new PropertyDescriptor("imageComment", beanClass, "getImageComment", null);

      imageIndex.setBound(true);
      imageComment.setBound(true);

      PropertyDescriptor properties[]
      = {imageNumber, imageIndex, imageComment};
      return properties;
   } catch (IntrospectionException e) {
       throw new Error(e.toString());
   }
 }
 public EventSetDescriptor[] getEventSetDescriptors() {
   try {
      EventSetDescriptor changed
      = new EventSetDescriptor(beanClass, "propertyChange", PropertyChangeListener.class, "propertyChange");
      changed.setDisplayName("Property Change");
      EventSetDescriptor events[] = {changed};
      return events;
   } catch (IntrospectionException e) {
      throw new Error(e.toString());
   }
 }
 public MethodDescriptor[] getMethodDescriptors() {
   try {
      MethodDescriptor showNext
      = new MethodDescriptor(beanClass.getMethod("showNext", null));

         MethodDescriptor methods[] = {showNext};
         return methods;
   } catch (Exception e) {
          throw new Error(e.toString());
   }
 }
 public java.awt.Image getIcon(int iconKind){
```

```
        if(iconKind == SimpleBeanInfo.ICON_COLOR_16x16)
          return loadImage("photoAlbumIcon16.gif");
        else
          return loadImage("photoAlbumIcon32.gif");
    }
  }
```

La classe viene definita come sottoclasse di SimpleBeanInfo, in modo da rendere il processo di sviluppo più rapido.

Il primo metodo, getPropertyDescriptors, restituisce un array con un tre PropertyDescriptor, uno per ciascuna delle proprietà che si vogliono rendere visibili. Il costruttore di PropertyDescriptor richiede quattro argomenti: il nome della proprietà, la classe del Bean, il nome del metodo getter e quello del metodo setter: quest'ultimo è posto a null, a significare che le proprietà sono di tipo Read Only. Si noti, in questo metodo e nei successivi, che la creazione dei Descriptors deve essere definita all'interno di un blocco try–catch, dal momento che può generare IntrospectionException.

Il secondo metodo, getEventSetDescriptors(), restituisce un vettore con un unico EventSetDescriptor. Quest'ultimo viene inizializzato con quattro parametri: la classe del Bean, il nome della proprietà, la classe dell'ascoltatore e la firma del metodo che riceve l'evento. Si noti la chiamata al metodo setDisplayName(), che permette di impostare un nome più leggibile di quello che viene normalmente ottenuto dalle firme dei metodi.

Il terzo metodo, getMethodDescriptors, restituisce un vettore contenente un unico MethodDescriptor, che descrive il metodo showNext(). Il costruttore di MethodDescriptor richiede come unico parametro un oggetto di classe Method, che in questo esempio viene richiesto alla classe PhotoAlbum ricorrendo alla reflection.

Infine il metodo getIcon() restituisce un'icona, che normalmente viene associata al Bean all'interno di strumenti visuali.

Per impacchettare il Bean PhotoAlbum con le icone e il BeanInfo, si può seguire la procedura già descritta, modificando la riga di comando dell'utility jar in modo da includere le icone nell'archivio.

```
jar cfm photoAlbum.jar photoAlbumManifest.tmp com\mokabyte\mokabook\
javaBeans\photoAlbum\*.class com\mokabyte\mokabook\javaBeans\
photoAlbum\*.gif com\mokabyte\mokabook\javaBeans\photoAlbum\images\*.*
```

## Personalizzazione dei Bean

L'aspetto e il comportamento di un Bean possono essere personalizzati in fase di composizione all'interno di un tool grafico abilitato ai Beans. Esistono due strumenti per personalizzare un Bean: gli Editor di proprietà e i Customizer. Gli Editor di proprietà sono componenti grafici specializzati nell'editing di un particolare *tipo* di proprietà: interi, stringhe, files… Ogni Editor di proprietà viene associato a un particolare tipo Java, e il tool grafico compone automaticamente un Property Sheet analizzando le proprietà di un Bean, e ricorrendo agli Editor più adatti alla circostanza. In fig. 18.6 si può vedere un esempio di Property Sheet, realizzato dal Bean Box: ogni riga presenta, accanto al nome della proprietà, il relativo Editor.

**Figura 18.6 –** *Un Property Sheet generato in modo automatico a partire dalle proprietà di un pulsante Bean*

**Figura 18.7 –** *Il Property Sheet relativo a un pulsante Bean. Si noti il pannello ausiliario FontEditor*

Un Customizer, d'altra parte, è un pannello di controllo specializzato per un particolare Bean: in questo caso è il programmatore a decidere cosa mostrare nel pannello e in quale maniera. Per questa ragione un Customizer viene associato, grazie al BeanInfo, a un particolare Bean e non può, in linea di massima, essere usato su Bean differenti.

## Come creare un Editor di proprietà

Un Editor di proprietà deve implementare l'interfaccia PropertyEditor, o in alternativa, estendere la classe PropertyEditorSupport che fornisce un'implementazione standard dei metodi dell'interfaccia. L'interfaccia PropertyEditor dispone di metodi che premettono di specificare come una proprietà debba essere rappresentata in un property sheet. Alcuni Editor consistono in uno strumento direttamente editabile, altri presentano uno strumento a scelta multipla, come un ComboBox; altri ancora, per permettere la modifica, aprono un pannello separato, come nella proprietà font dell'esempio, che viene modificata grazie al pannello ausiliario FontEditor.

Per fornire il supporto a queste modalità di editing, bisogna implementare alcuni metodi di PropertyEditor, in modo che ritornino valori non nulli.

I valori numerici o String possono implementare il metodo setAsText(String s), che estrae il valore della proprietà dalla stringa che costituisce il parametro. Questo sistema permette di inserire una proprietà con un normale campo di testo.

Gli Editor standard per le proprietà Color e Font usano un pannello separato, e ricorrono al Property Sheet solamente per mostrare l'impostazione corrente. Facendo click sul valore, viene aperto l'Editor vero e proprio. Per mostrare il valore corrente della proprietà, è necessario sovrascrivere il metodo isPaintable() in modo che restituisca true, e sovrascrivere paintValue in modo che dipinga la proprietà attuale in un rettangolo all'interno del Property Sheet.

Per supportare l'Editor di Proprietà personalizzato occorre sovrascrivere altri due metodi della classe PropertyEditorSupport: supportsCustomEditor, che in questo caso deve restituire true, e getCustomEditor, in modo che restituisca un'istanza dell'Editor.

## Registrare gli Editor

I Property Editor vengono associati alle proprietà attraverso un'associazione esplicita, all'interno del metodo getPropertyDescriptors() del BeanInfo, con una chiamata al metodo setPropertyEditorClass(Class propertyEditorClass) del PropertyDescriptor corrispondente, come avviene nel Bean Molecule

```
PropertyDescriptor pd = new PropertyDescriptor("moleculeName", Molecule.class);
pd.setPropertyEditorClass(MoleculeNameEditor.class);
```

**Figura 18.8 –** *Il Bean Molecule associa alla proprietà moleculeName di un Editor di proprietà personalizzato*

In alternativa si può registrare l'Editor con il seguente metodo statico

```
PropertyEditorManager.registerEditor(Class targetType, Class editorType)
```

che richiede come parametri la classe che specifica il tipo e quella che specifica l'Editor.

## Customizers

Con un Bean Customizer è possibile fornire un controllo completo sul modo in cui configurare ed editare un Bean. Un Customizer è in pratica una piccola applicazione specializzata nell'editing di un particolare Bean, ogni volta che la configurazione di un Bean richiede modalità troppo sofisticate per il normale processo di creazione automatica del Property Sheet.

Le uniche regole a cui ci si deve attenere per realizzare un Customizer sono:

- deve estendere la classe `Component`, o una delle sue sottoclassi;
- deve implementare l'interfaccia `java.bean.Customizer`;
- deve implementare un costruttore privo di parametri.

Per associare il Customizer al proprio Bean, bisogna sovrascrivere il metodo `getBeanDescriptor` nella classe `BeanInfo`, in modo che restituisca un opportuno `BeanDescriptor`, il quale a sua volta dovrà restituire la classe del Customizer alla chiamata del metodo `getCustomizerClass`.

## Serializzazione

Per rendere serializzabile una classe Bean è di norma sufficiente implementare l'interfaccia Serializable, sfruttando così l'Object Serialization di Java. L'interfaccia Serializable non contiene metodi: essa viene usata dal compilatore per marcare le classi che possono essere serializzate. Esistono solo poche regole per implementare classi Serializable: anzitutto è necessario dichiarare un costruttore privo di argomenti, che verrà chiamato quando l'oggetto verrà ricostruito a partire da un file .ser; in secondo luogo una classe serializzabile deve definire al suo interno solamente attributi serializzabili.

Se si desidera fare in modo che un particolare attributo non venga salvato al momento della serializzazione, si può ricorrere al modificatore transient. La serializzazione standard, inoltre, non salva lo stato delle variabili static.

Per tutti i casi in cui la serializzazione standard non risultasse applicabile, occorre procedere all'implementazione dell'interfaccia Externalizable, fornendo, attraverso i metodi readExternal(ObjectInput in) e writeExternal(ObjectOutput out), delle istruzioni esplicite su come salvare lo stato di un oggetto su uno stream e come ripristinarlo in un secondo tempo.

# Installazione dell'SDK

GIOVANNI PULITI

## Scelta del giusto SDK

Per poter lavorare con applicazioni Java o crearne di nuove, il programmatore deve poter disporre di un ambiente di sviluppo e di esecuzione compatibile con lo standard 100% Pure Java. Sun da sempre rilascia un kit di sviluppo che contiene tutti gli strumenti necessari per la compilazione ed esecuzione di applicazioni Java. Tale kit è comunemente noto come Java Development Kit (JDK): nel corso del tempo sono state rilasciate le versioni 1.0, 1.1, 1.2, 1.3 e 1.4. Attualmente l'ultima versione rilasciata è la 1.4, mentre si annuncia un prossimo JDK 1.5. Il JDK comprende una Java Virtual Machine (JVM), invocabile con il comando `java`, un compilatore (comando `javac`), un debugger (`jdbg`), un interprete per le applet (`appletviewer`) e altro ancora.

A partire dalla versione 1.2, Sun ha introdotto una nomenclatura differente per le varie versioni del kit di sviluppo. In quel momento nasceva infatti Java 2, a indicare la raggiunta maturità del linguaggio e della piattaforma. Pur mantenendo la completa compatibilità con il passato, Java 2 ha introdotto importanti miglioramenti, quali una maggiore stabilità e sicurezza, migliori performance e l'ottimizzazione dell'uso della memoria.

Con Java 2 nasce il concetto di SDK: non più un Java Development Kit ma un Software Development Kit. Il linguaggio Java può essere finalmente considerato un potente strumento general purpose.

La notazione di JDK non è stata eliminata: il JDK è formalmente una release dell'SDK Java 2.

Con Java 2, per organizzare e raccogliere al meglio le diverse tecnologie che costituiscono ormai la piattaforma, Sun ha suddiviso l'SDK in tre grandi categorie:

- Java 2 Standard Edition (J2SE): questa versione contiene la JVM standard più tutte le librerie necessarie per lo sviluppo della maggior parte delle applicazioni Java.

- Java 2 Enterprise Edition (J2EE): contiene in genere le API enterprise come EJB, JDBC 2.0, Servlet ecc. La JVM normalmente è la stessa, quindi lavorare direttamente con l'SDK in boundle spesso non è molto utile: è molto meglio partire dalla versione J2SE e aggiungere l'ultima versione delle API EE, a seconda delle proprie esigenze.

- Java 2 Micro Edition (J2ME): Java è nato come linguaggio portabile in grado di essere eseguito con ogni tipo di dispositivo. La J2ME include una JVM e un set di API e librerie appositamente limitate, per poter essere eseguite su piccoli dispositivi embedded, telefoni cellulari ed altro ancora. Questa configurazione deve essere scelta solo se si vogliono scrivere applicazioni per questo genere di dispositivi.

La procedura di installazione è in genere molto semplice, anche se sono necessarie alcuni piccoli accorgimenti per permettere un corretto funzionamento della JVM e dei programmi Java. Si limiterà l'attenzione alla distribuzione J2SE. Per chi fosse interessato a scrivere programmi J2EE non vi sono particolari operazioni aggiuntive da svolgere. Per la J2ME, invece, il processo è del tutto analogo, anche se si deve seguire un procedimento particolare a seconda del dispositivo scelto e della versione utilizzata.

I file per l'installazione possono essere trovati direttamente sul sito di Sun, come indicato in [SDK]. Altri produttori rilasciano JVM per piattaforme particolari (molto note e apprezzate sono quelle di IBM). Per la scelta di JVM diverse da quelle prodotte da Sun si possono seguire le indicazioni della casa produttrice o del particolare sistema operativo utilizzato.

Chi non fosse interessato a sviluppare applicazioni Java, ma solo a eseguire applicazioni già finite, potrà scaricare al posto dell'SDK il Java Runtime Environment (JRE) , che in genere segue le stesse edizioni e release dell'SDK. Non sempre il JRE è sufficiente: per esempio, se si volessero eseguire applicazioni JSP già pronte, potrebbe essere necessario mettere ugualmente a disposizione di Tomcat (o di un altro servlet-JSP engine) un compilatore Java, indispensabile per la compilazione delle pagine JSP.

# Installazione su Windows

In ambiente Windows, in genere, il file di installazione è un eseguibile autoinstallante, che guida l'utente nelle varie fasi della procedura.

Non ci sono particolari aspetti da tenere in considerazione, a parte la directory di installazione e le variabili d'ambiente da configurare. Per la prima questione, a volte l'installazione nella directory "program files" può causare problemi di esecuzione ad alcuni applicativi Java che utilizzano la JVM di sistema (per esempio Tomcat o JBoss). Per questo, si consiglia di installare in una directory con un nome unico e senza spazi o altri caratteri speciali ("c:\programs", "c:\programmi" o semplicemente "c:\java").

Per poter funzionare, una qualsiasi applicazione Java deve sapere dove è installato il JDK, e quindi conoscere il path dell'eseguibile java (l'interprete), di javac (il compilatore usato da Tomcat per compilare le pagine JSP) e di altri programmi inclusi nel JDK.

Inoltre, un programma Java deve anche poter ricavare la variabile d'ambiente CLASSPATH, all'interno della quale dovranno essere inseriti i riferimenti ai vari package utilizzati (directory scompattate, file .jar o .zip). Al momento dell'installazione, il classpath viene automaticamente impostato in modo da contenere le librerie di base del Java SDK (in genere, nella sottodirectory jre/lib, o semplicemente lib).

A partire dal JDK 1.1 è invalsa l'abitudine di utilizzare la variabile JAVA_HOME, che deve puntare alla directory di installazione del JDK. Di conseguenza, il path di sistema dovrà essere impostato in modo che punti alla directory %JAVA_HOME%\bin.

Di norma, queste impostazioni sono effettuate in modo automatico dal programma di installazione, ma possono essere facilmente modificate o impostate ex-novo tramite il pannello di controllo di Windows.

Per esempio, aprendo la finestra per l'impostazione delle variabili d'ambiente dal pannello di controllo...

**Figura A.1 –** *In Windows il pannello di controllo permette di impostare le diverse variabili d'ambiente.*

...si può procedere a inserire sia la variabile JAVA_HOME (in questo caso `c:\programs\jdk1.4`) sia la directory con gli eseguibili nel path (`%JAVA_HOME%\bin`).

**Figura A.2 –** *Come impostare la variabile JAVA_HOME in modo che punti alla directory di installazione del JDK.*

**Figura A.3 –** *Come impostare il path in modo che includa la dir* JAVA_HOME\bin.

Se tutto è stato fatto come si deve, aprendo una console DOS si può verificare la correttezza delle impostazioni inserite.

Per esempio, per conoscere il contenuto della variabile JAVA_HOME si potrà scrivere:

```
C:\>echo %JAVA_HOME%
C:\programs\jdk1.4
```

Il comando path, invece, mostrerà Tra le altre cose:

```
C:\> path
PATH=…….C:\programs\jdk1.4\bin
```

A questo punto, si può provare a eseguire la JVM con il comando:

```
C:\>java -version
```

L'opzione –version permette di conoscere la versione della JVM installata. In questo caso, il comando restituisce il seguente output:

```
java version "1.4.1"
Java(TM) 2 Runtime Environment, Standard Edition (build 1.4.1-b21)
Java HotSpot(TM) Client VM (build 1.4.1-b21, mixed mode)
```

se si usa un sistema basato su Windows 95-98 o ME, l'impostazione delle variabili d'ambiente può essere fatta tramite il file autoexec.bat. In questo caso, con un'istruzione del tipo:

set JAVA_HOME="…."

Si potrà definire la variabile in modo che punti alla directory indicata. Si tenga presente che tali sistemi operativi offrono un supporto ridotto per lo sviluppo e l'esecuzione di applicazioni che fanno uso dei protocolli di rete (socket TCP, database ecc...) o server side (servlet, web application, EJB, per esempio). Si consiglia pertanto di utilizzare le versioni più evolute (NT, 2000, XP), che supportano in modo più corretto e completo lo strato di network TCP/IP e offrono maggiori servizi.

# Installazione su Linux

Per l'installazione su Linux, la procedura è molto simile a quella per Windows: si deve scaricare un file di installazione e installarlo. Per quest'ultimo aspetto si può utilizzare un rpm autoinstallante o un file eseguibile con estensione .bin.

Per esempio, se j2sdk-1.4.2-nb-3.5-bin-linux.bin è il file installante scaricato dal sito Sun, per prima cosa lo si renda eseguibile con il comando:

```
chmod o+x j2sdk-1.4.2-nb-3.5-bin-linux-i586.bin
```

quindi lo si mandi in esecuzione tramite:

```
./j2sdk-1.4.2-nb-3.5-bin-linux-i586.bin
```

per eseguire l'installazione. Di norma, questo porterà all'installazione dell'SDK in una directory il cui nome segue lo schema usr/java/jdk-<version-number>.

Questo significa che dovranno essere modificate di conseguenza le variabili JAVA_HOME e PATH, intervenendo sui file di profilo .bashrc o .bash_properties (a seconda del tipo di shell usata) dell'utente che dovrà usare Java:

```
JAVA_HOME=/usr/java/jdk1.4.1/
export JAVA_HOME
PATH=$JAVA_HOME/bin:$PATH
export PATH
```

Nel caso in cui un'applicazione debba far uso di altri package oltre a quelli di base del Java SDK, come un parser XML Xerces (contenuto in genere in un file xerces.jar), il package Java Mail, la Servlet API o altro ancora, si dovrà aggiungere manualmente al classpath il contenuto di tali librerie. Questo può essere fatto in due modi.

Il primo sistema consiste nell'aggiungere tali librerie al classpath di sistema, tramite il pannello di controllo di Windows o mediante l'impostazione ed esportazione di una variabile globale su Linux. In questo caso si potrà essere sicuri che tutte le applicazioni che dovranno utilizzare un parser Xerces o JavaMail potranno funzionare correttamente senza ulteriori impostazioni.

Attualmente, però, lo scenario Java è molto complesso, quindi un'impostazione globale difficilmente si adatta a tutte le applicazioni: in un caso potrebbe essere necessaria la versione 1.0 di Xerces, mentre un'altra applicazione potrebbe funzionare solo con la 1.2. Per questo motivo, in genere, si preferisce impostare un classpath personalizzato per ogni applicazione, passando tale configurazione alla JVM con il flag –classpath o –cp. Per esempio, in Windows si potrebbe scrivere:

```
set MY_CP=c:\programs\mokabyte\mypackages.jar
java –cp %MY_CP% com.mokabyte.mokacode.TestClasspathApp
```

Dove TestClasspathApp potrebbe essere un'applicazione che abbia bisogno di una serie di classi e interfacce contenute in mypackages.jar.

In questo modo si potranno costruire tutti i classpath personalizzati, concatenando file e directory di vario tipo.

In ambito J2EE le cose si complicano: entrano infatti in gioco il tipo di applicazione e le regole di caricamento del classloader utilizzato. Per questi aspetti, che comunque riguardano il programmatore esperto, si rimanda alla documentazione del prodotto utilizzato, e si consiglia l'adeguamento alle varie convenzioni imposte dalla specifica Java.

# Bibliografia

[SUN] – Sito web ufficiale di Sun dedicato a Java: http://java.sun.com

[SDK] – Sito web di Sun per il download dell'SDK nelle versioni per le varie piattaforme: http://java.sun.com/downloads

[JIBM] – Sito web IBM dedicato a Java: http://www.ibm.com/java

[xIBM] – "IBM Developer Kit for Linux: Overview": http://www-106.ibm.com/developerworks/java/jdk/linux140/?dwzone=java

# Ginipad, un ambiente di sviluppo per principianti

ANDREA GINI

Ginipad è un ambiente di sviluppo per Java realizzato da Andrea Gini, uno degli autori di questo manuale. Ginipad è stato pensato come strumento per principianti, che non hanno tempo o voglia di barcamenarsi tra editor testuali e tool a riga di comando. La sua interfaccia grafica semplice ed essenziale ne ha decretato il successo anche presso utenti più esperti, che spesso necessitano di uno strumento rapido e leggero da alternare agli ambienti di sviluppo più complessi.

Ginipad è stato progettato per offrire il massimo grado di funzionalità nel modo più semplice e intuitivo possibile. Bastano cinque minuti per prendere confidenza con l'ambiente e le sue funzioni. Questa appendice fornisce una tabella riassuntiva dei principali comandi e una guida all'installazione.

Si consiglia di visitare la home page del progetto per trovare tutte le informazioni necessarie:

http://www.mokabyte.it/ginipad

Il tempo di apprendimento può essere ridotto ad appena cinque minuti grazie a uno slideshow in PowerPoint, disponibile all'indirizzo:

http://www.mokabyte.it/ginipad/download/GinipadVisualTutorial.ppt

# Caratteristiche principali

**Figura B.1 –** *Caratteristiche principali di Ginipad.*

1. Pochi pulsanti facili da identificare.
2. Possibilità di lavorare su più di un documento.
3. Editor con Syntax Highlight.
4. Indice navigabile di metodi, campi e classi interne.
5. Autocompletamento delle dichiarazioni.
6. Hyperlink verso gli errori.

# Tabella riassuntiva dei comandi

| File | | | |
|---|---|---|---|
| | New | (Ctrl-N) | Crea un nuovo sorgente Java. |
| | Open | (Ctrl-O) | Carica un sorgente da disco. |
| | Save As | (Ctrl-S) | Salva il documento corrente. |
| | Close | (Ctrl-W) | Chiude il documento corrente. |
| | Open All | | Apre in una volta sola gli ultimi otto sorgenti. |
| | Exit | | Chiude il programma. |
| **Edit** | | | |
| | Cut | (Ctrl-X) | Taglia il testo selezionato. |
| | Copy | (Ctrl-C) | Copia il testo selezionato. |
| | Paste | (Ctrl-V) | Incolla il testo contenuto nella clipboard. |
| | Select All | (Ctrl-A) | Seleziona tutto il contenuto dell'editor. |
| | Undo | (Ctrl-Z) | Annulla l'ultima modifica. |
| | Redo | (Ctrl-Y) | Ripristina l'ultima modifica. |
| | Find | (Ctrl-F) | Apre la finestra di dialogo Find. |
| | Replace | (Ctrl-R) | Apre la finestra di dialogo Replace. |
| **Tools** | | | |
| | Compile | (Ctrl-Shift-C) | Compila il documento corrente. |
| | Run | (Ctrl-Shift-R) | Esegue il documento corrente. |
| | Stop | | Interrompe l'esecuzione del processo corrente. |
| | Format source code | (Ctrl-Shift-F) | Esegue una formattazione del codice. |
| **Console** | | | |
| | Hide Tree | | Nasconde il componente ad albero. |
| | Show Tree | | Mostra il componente ad albero. |
| | Hide Console | | Nasconde la console. |
| | Show Console | | Mostra la console. |
| | Show Panels | | Mostra tutti i pannelli. |

| | Full Screen | | Espande l'editor a pieno schermo. |
|---|---|---|---|
| | Clear Console | | Ripulisce la console. |
| **Dialog** | | | |
| | Preferences | | Apre la finestra delle preferenze. |
| | Help | | Apre la finestra di Help. |
| | About | | Apre la finestra di About. |

# Installazione

1. Caricare e Installare il JDK 1.4.
2. Lanciare il file di setup di Ginipad.
3. Al secondo passaggio della fase di installazione verrà richiesto di scegliere la Virtual Machine. Si consiglia di scegliere quella presente nella cartella \bin del JDK, come si vede in Fig B.2.

**Figura B.2 –** *Scelta della Virtual Machine.*

4. Al termine dell'installazione si può avviare il programma. Al primo avvio, Ginipad effettua una ricerca per localizzare la posizione del JDK. Tale processo è automatico e trasparente all'utente.

## Cosa fare se Ginipad non trova il JDK

Ginipad è in grado di identificare da solo la posizione del JDK su disco, durante la fase di installazione. Tuttavia, se tale posizione dovesse cambiare, per esempio in seguito a un aggiornamento del JDK, all'avvio successivo verrà richiesto di indicare la nuova posizione dell'ambiente di sviluppo.

**Figura B.3 -** *La finestra per aprire il File Chooser.*

Dopo aver dato l'OK, verrà visualizzata una finestra File Chooser, tramite la quale si dovrà localizzare la directory del JDK sul disco. Una volta trovata la cartella, non resta che premere il pulsante Locate JDK Directory.

# Parole chiave

ANDREA GINI

| abstract | double | int | super |
|---|---|---|---|
| boolean | else | interface | switch |
| break | extends | long | synchronized |
| byte | final | native | this |
| case | float | package | throw |
| char | for | private | throws |
| class | (goto) | protected | transient |
| (const) | if | public | try |
| continue | implements | return | void |
| default | import | short | volatile |
| do | instanceof | static | while |

La maggior delle parole riservate di Java deriva dal C, il linguaggio dal quale Java ha ereditato la sintassi delle strutture di controllo. Le somiglianze con il C++, al contrario, sono minori, dal momento che Java adotta una sintassi differente per quanto riguarda i costrutti caratteristici della programmazione a oggetti. Le parole chiave `goto` e `const`, presenti nel C, fanno parte dell'insieme delle keyword, ma di fatto non compaiono nel linguaggio Java: in questo modo, il compilatore può segnalare uno speciale messaggio di errore se il programmatore le dovesse utilizzare inavvertitamente.

Appendice D

# Diagrammi di classe e sistemi orientati agli oggetti

Andrea Gini

Un effetto della strategia di incapsulamento è quello di spingere il programmatore a esprimere il comportamento di un sistema a oggetti unicamente attraverso l'interfaccia di programmazione delle classi. In questo senso, quando un programmatore si trova a dover utilizzare una libreria di classi realizzate da qualcun altro, non è interessato a come essa sia stata effettivamente implementata: di norma, è sufficiente conoscere le firme dei metodi, le relazioni di parentela tra le classi, le associazioni e le dipendenze, informazioni che non dipendono dall'implementazione dei singoli metodi.

Il diagramma di classe è un formalismo che permette di rappresentare per via grafica tutte queste informazioni, nascondendo nel contempo i dettagli di livello inferiore. L'uso dei diagrammi di classe permette di vedere un insieme di classi Java da una prospettiva più alta rispetto a quella fornita dal codice sorgente, simile a quella che si ha quando si guarda una piantina per vedere com'è fatta una città. La piantina non contiene tutti i dettagli della zona rappresentata, come la posizione delle singole abitazioni o dei negozi, ma riporta informazioni sufficienti per orientarsi con precisione.

I diagrammi di classe fanno parte di UML (Unified Modeling Language), un insieme di notazioni grafiche che permette di fornire una rappresentazione dei diversi aspetti di un sistema software orientato agli oggetti, indipendentemente dal linguaggio di programmazione effettivamente utilizzato. L'UML comprende sette tipi diversi di diagrammi, che permettono di modellare i vari aspetti dell'architettura e del comportamento di un sistema software prima di iniziarne lo sviluppo. I diagrammi UML costituiscono una parte fondamentale della documentazione di un sistema informativo, e forniscono una guida essenziale in fase di studio o di manutenzione del sistema stesso.

L'UML non è un linguaggio di programmazione, anche se negli ultimi anni gli ambienti di sviluppo hanno iniziato a includere strumenti che permettono di produrre codice a partire dai

diagrammi e viceversa. I seguenti paragrafi vogliono fornire una guida essenziale ai diagrammi di classe, l'unico formalismo UML presente in questo libro.

# Classi e interfacce UML

In UML le classi e le interfacce sono rappresentate come rettangoli, suddivisi in tre aree: l'area superiore contiene il nome della classe o dell'interfaccia, quella intermedia l'elenco degli attributi e quella inferiore l'elenco dei metodi:

**Figura D.1 –** *Un esempio di classe e di interfaccia in UML.*

Entrambi i diagrammi non contengono alcun dettaglio sul contenuto dei metodi: il comportamento di una classe o di un'interfaccia UML è espresso unicamente tramite il nome dei suoi metodi. Le firme di metodi e attributi seguono una convenzione differente rispetto a quella adottata in Java: in questo caso, il nome precede il tipo, e tra i due compare un simbolo di due punti (:) come separatore. I parametri dei metodi, quando presenti, seguono la stessa convenzione. Il simbolo più (+), presente all'inizio, denota un modificatore public, mentre il trattino (-) indica private e il cancelletto (#) significa protected.

Il diagramma di interfaccia presenta alcune differenze rispetto a quello di classe:

- Al di sopra del nome compare un'etichetta "interface".
- Gli attributi (normalmente assenti) sono sottolineati, a indicare che si tratta di attributi statici immutabili.
- I metodi sono scritti in corsivo, per indicare che sono privi di implementazione.

Si osservi una tipica implementazione Java del diagramma di classe presente in figura 1:

```
public class SchedaAnagrafica {
```

```
    private String nome;
    private String cognome;

    public String getNome() {
      return nome;
    }
    public void setNome(String nome) {
            this.nome = nome;
    }
    public String getCognome() {
      return cognome;
    }
    public void setCognome(String cognome) {
      this.cognome = cognome;
    }
}
```

Spesso il diagramma di classe presenta un livello di dettaglio inferiore rispetto al codice sottostante: tipicamente, si usa un diagramma per descrivere un particolare aspetto di una classe, e si omettono i metodi e gli attributi che non concorrono a definire tale comportamento. In questo libro, i diagrammi di classe sono stati disegnati secondo questa convenzione.

# Ereditarietà e realizzazione

L'ereditarietà è rappresentata in UML con una freccia dalla punta triangolare, che parte dalla classe figlia e punta alla classe padre:

**Figura D.2 –** *Ereditarietà tra le classi.*

La realizzazione, equivalente all'implementazione di un'interfaccia in Java, viene rappresentata con una freccia simile a quella usata per l'ereditarietà, ma tratteggiata. Si noti che si ricorre alla realizzazione solo quando una classe implementa un'interfaccia, mentre se un'interfaccia ne estende un'altra si utilizza la normale ereditarietà.

In figura D.3 è possibile vedere un diagramma di classe contenente una relazione di ereditarietà tra interfacce (l'interfaccia MediaRecorder è figlia dell'interfaccia MediaPlayer) e due casi di realizzazione (la classe LettoreDVD realizza l'interfaccia MediaPlayer, mentre la classe VideoRegistratore realizza MediaRecorder).

**Figura D.3 –** *Un diagramma che contiene sia l'ereditarietà sia la realizzazione.*

# Associazione

L'associazione, rappresentata da una linea che congiunge due classi, denota una relazione di possesso. Un'associazione può essere bidirezionale o unidirezionale. Nel secondo caso, al posto di una linea semplice si utilizza una freccia. La freccia indica la direzione del flusso della comunicazione: in pratica, la classe da cui parte la freccia può chiamare i metodi di quella indicata dalla punta, ma non viceversa. L'equivalente Java dell'associazione è la presenza di un attributo in una classe, che di fatto denota il possesso di un particolare oggetto e la possibilità di invocare metodi su di esso.

**Figura D.4 –** *Classi unite da associazioni.*

In figura D.4 è possibile osservare un insieme di classi caratterizzate da associazioni sia uni- sia bidirezionali: un computer è collegato alle schede audio e video da associazioni bi direzionali, a indicare che la comunicazione avviene in entrambe le direzioni; le due schede, invece, presentano un'associazione unidirezionale rispettivamente con gli speaker e il monitor, poiché non è permessa la comunicazione in senso inverso.

# Aggregazione

Un tipo speciale di associazione è l'aggregazione, rappresentata da una linea tra due classi con un'estremità a diamante, che denota un'associazione uno a molti. In figura D.5 si può osservare una relazione uno a molti tra una biblioteca e i libri in essa contenuti.

**Figura D.5 –** *Un esempio di aggregazione.*

A parte la cardinalità, l'aggregazione equivale a un'associazione: nell'esempio di figura D.5 la classe Biblioteca possiede una collezione di libri e può invocare metodi su ognuno di essi. In Java, l'aggregazione corrisponde solitamente a un attributo di tipo `Vector` o `HashTable`, o più semplicemente a un array.

# Dipendenza

La dipendenza è rappresentata da una freccia tratteggiata. Letteralmente, la dipendenza suggerisce che la classe puntata dalla freccia esista indipendentemente dalla classe da cui parte la freccia: in Java, questo significa che la prima può essere compilata e utilizzata anche in assenza della seconda.

**Figura D.6 –** *Relazione di dipendenza in UML.*

La figura D.6 presenta un esempio di relazione di dipendenza: il CD, inteso come supporto per musica e dati, esiste indipendentemente dal particolare lettore con cui lo si legge: le sue caratteristiche sono definite da un documento denominato Red Book, al quale si devono attenere i produttori di lettori CD. Si noti l'etichetta "play" che compare sopra la freccia: le etichette permettono di fornire maggiori informazioni sul tipo di relazione che sussiste tra due classi.

# Indice analitico

## C



## D

## J

## K



## L



## M

## N



## O



## P



## R



## S

## T



## U



## V

# X